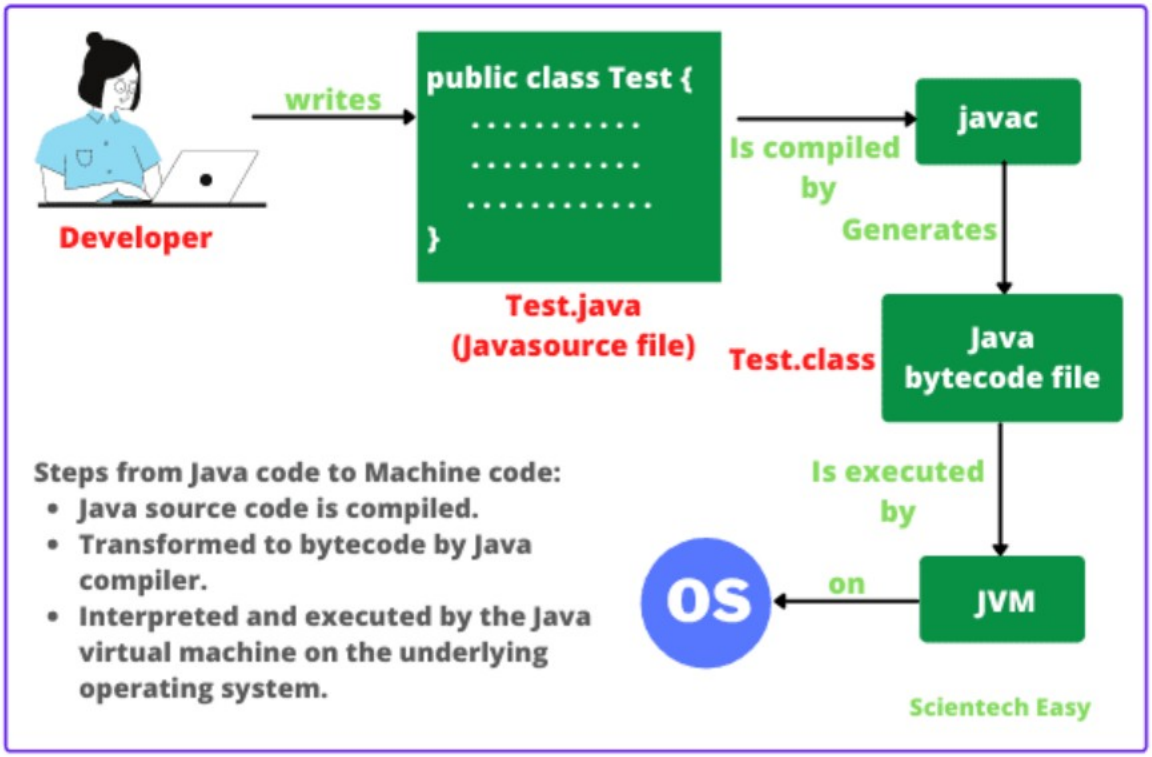
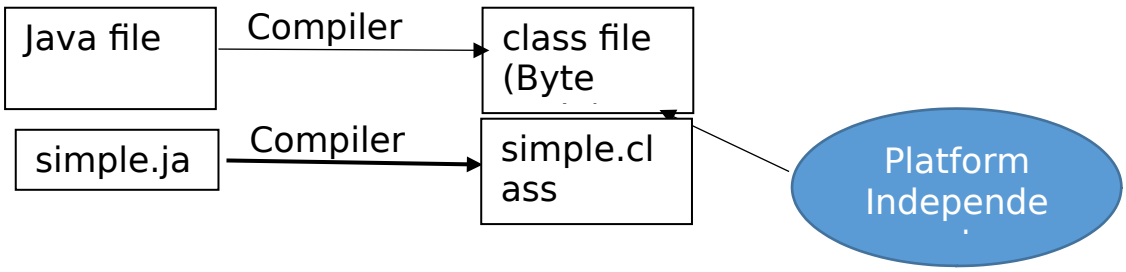
Java Tutorial

Course Outline:

1. Platform Independent:





javac – java compiler → used to compile java program

Please Join in below link

Instagram: https://www.instagram.com/rba_infotech/
LinkedIn: https://www.linkedin.com/company/rba-infotech/
Facebook: https://www.facebook.com/people/RBA-Infotech/100043552406579/

Syntax:

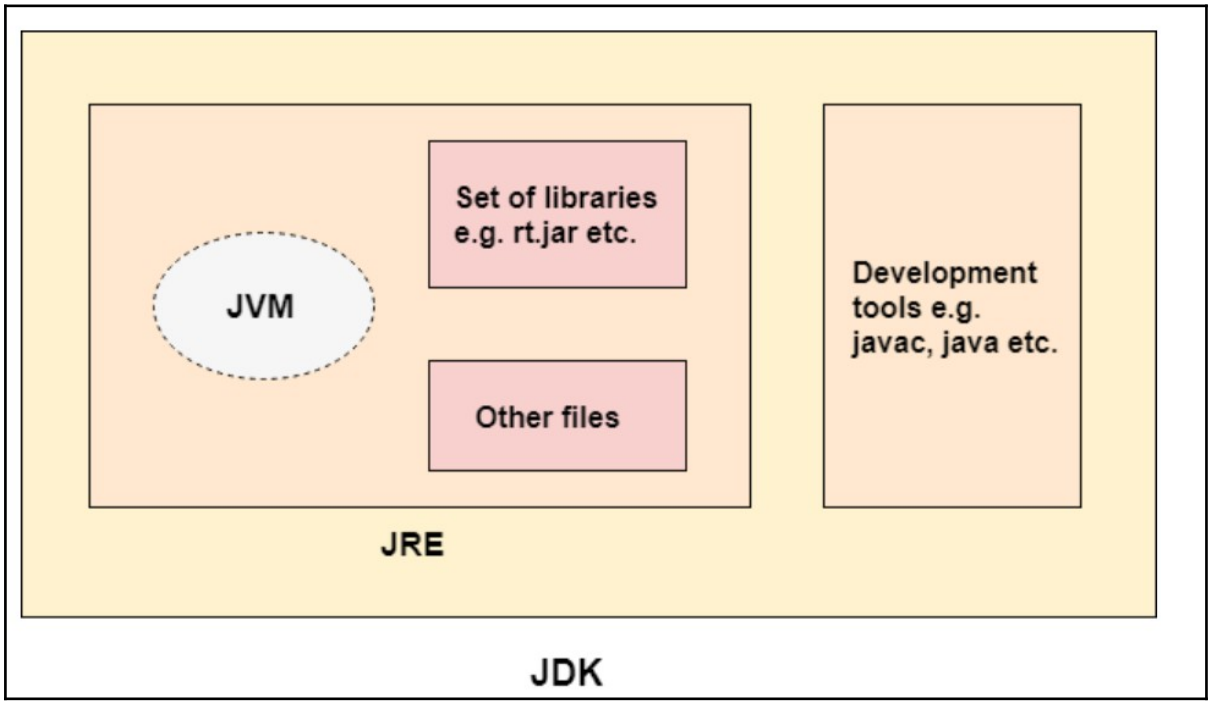
javac sample.java

java → Interpreter → used to run the java program

Syntax:

java sample

2. JDK vs JRE vs JVM



3. Setting Class Path

1. Using the -classpath or -cp Option (Preferred)

Syntax:

javac -cp <classpath> ClassName.java

Ex:

Javac -cp c:/lib/mylibrary.jar MyApp.java

Syntax:

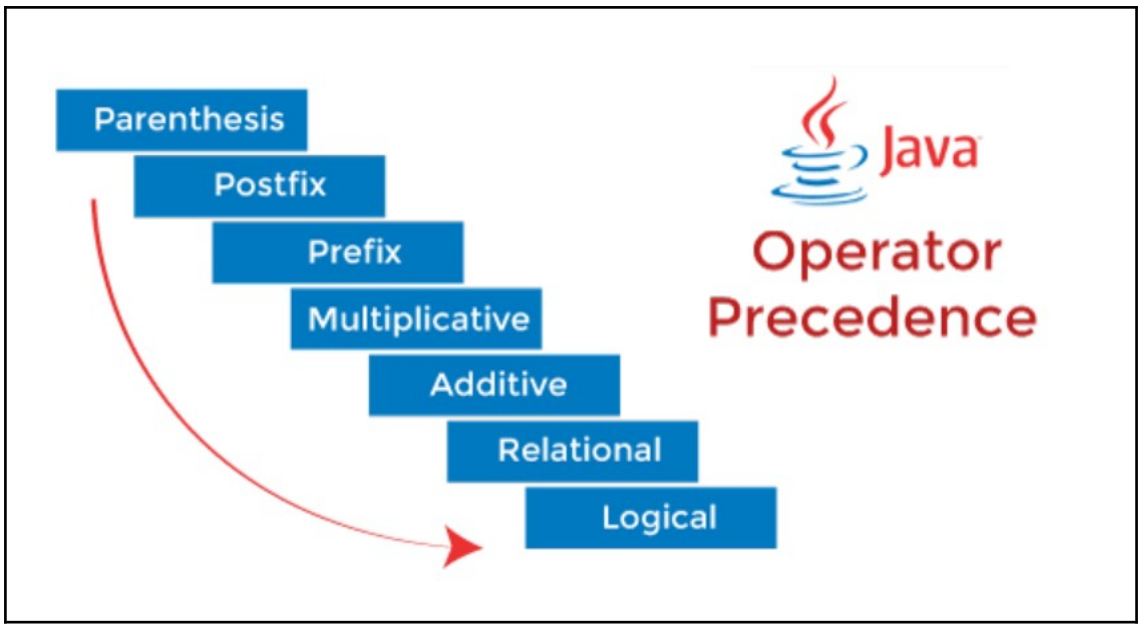
java -cp <classpath> ClassName

Ex:

java -cp c:lib/mylibrary.jar com.example.MyApp

1. Using the CLASSPATH Environment Variable

4. Operator Precedence



Level	Operator	Description	Associativity
16	() [] new .	parentheses array access object creation member access	left-to-right

	::	method reference	
15	++ --	unary post-increment unary post-decrement	left-to-right
14	+ - ! ~ ++ --	unary plus unary minus unary logical NOT unary bitwise NOT unary pre-increment unary pre-decrement	right-to-left
13	()	cast	right-to-left
12	* / %	multiplicative	left-to-right
11	+ - +	additive string concatenation	left-to-right
10	<< >> >>>	shift	left-to-right
9	< <= > >= instanceof	relational	left-to-right
8	== !=	equality	left-to-right
7	&	bitwise AND	left-to-right
6	^	bitwise XOR	left-to-right
5	|	bitwise OR	left-to-right
4	&&	logical AND	left-to-right
3	||	logical OR	left-to-right
2	?:	ternary	right-to-left

1	= += -= *= /= %= &= ^= |= <<= >>= >>>=	assignment	right-to-left
0	-> ->	lambda expression switch expression	right-to-left

Eclipse – IDE (Integrated Development Environment)

5. Class

1. User Defined Class
2. System Defined Class / Predefined Class (Ex. Scanner, System)

Variable

Data Types

Operators

Expression

Keywords

6. Practice the below programs:

https://www.youtube.com/playlist?list=PLqSczVKioQu71VOiW6BBXEW-BaAwBbHUf

1 byte = 8 bit

+	1	1	1	1	1	1	1

/ -							

$1 * 2^0 = 1 + 2 + 4 + 8 + 16 + 32 + 64 = 127$

0 or 1

7. Control Statements / Structures:

1. Decision Making Statements

a. If
b. If .. else
c. If ...else if
d. If ...else if ...else if ...else
e. Nested if

2. Loop / Iterative Statements

a. While
b. For loop
c. For each or Enhanced For loop(will be explained with Array)
d. Do while
e.

3. Jump Statements

f. Break – Execution of flow comes out of the loop
g. Continue – transfer the flow of execution to the starting of the loop

4. Switch Case Loop

h. Switch .... Case statement

8. Array:

Is a collection of similar datatypes value that can store values in sequential order.

Array is Fixed Size which is drawback of array.

[] -> Array

1. Single Dimensional Array

Declaration:

Ex.1:

int[] rollNo = new int[10];

or

int rollNo[] = new int[10];

Or

Int []rollNo = new int[10];

or

int[] rollNo;

rollNo = new int[10];

Initialize the array variables

int[] numbers = {5, 3, 8, 1, 2};

Int[] rollNo = new int[10]; 10 * 4 = 40 bytes – Static Memory allocation

45	21	52	61	90	76	21	33	73	54

Index -> 0 1 2 3 4 5 6 7 8 9

Please Join in below link

Instagram: https://www.instagram.com/rba_infotech/
LinkedIn: https://www.linkedin.com/company/rba-infotech/
Facebook: https://www.facebook.com/people/RBA-Infotech/100043552406579/

rollNo[0] = 45

rollNo[1] = 21

.

.

rollNo[9] = 88

2. Two Dimensional Array

Int[][] arr = new int[3][5];

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2		
2.0				

arr[0][0] = 15;

arr[0][1] = 74;

arr[][]

9. String

String – is a collection of characters.

Delcaration:

Ex 1:

String s1=”SELENIUM”;

String s2 = “SELENIUM”;

String s1;

s1=”Selenium” Memory Address for S1 = 1000

Or

String s3 = new String(“SELENIUM”); New Memory Address for S2 = 2000

S	E	L	E	N	I	U	M	\0
0	1	2	3		4	5	6	7

Senthil is good - 13

S	E	N	T	H	I	L		I	S		G	O	O	D	\0

10. Java Naming Conventions

Java naming convention is the guideline to follow when giving name for identifiers such as class, object, method, variable, constant, package, interface etc.

11. Different types of Variables in Java

Variable Type	Declared In	Scope	Stored In	Default Value
Local	Inside method	Method/block only	Stack	None (must init)
Instance	Inside class	Whole object	Heap	Yes
Static	Inside class	Entire class	Method area	Yes
Parameter	Method signature	Method/block only	Stack	Must be passed

12. Java OOPs Concepts (Object Oriented Programming)

1. Access Modifier / Access Specifier

2. Class
3. Object
4. Encapsulation
5. Inheritance
6. Polymorphism
7. Abstraction
    a. Abstract Class
    b. Interface
8. Constructor
9. Package
10. Keyrords: this, static, super, final

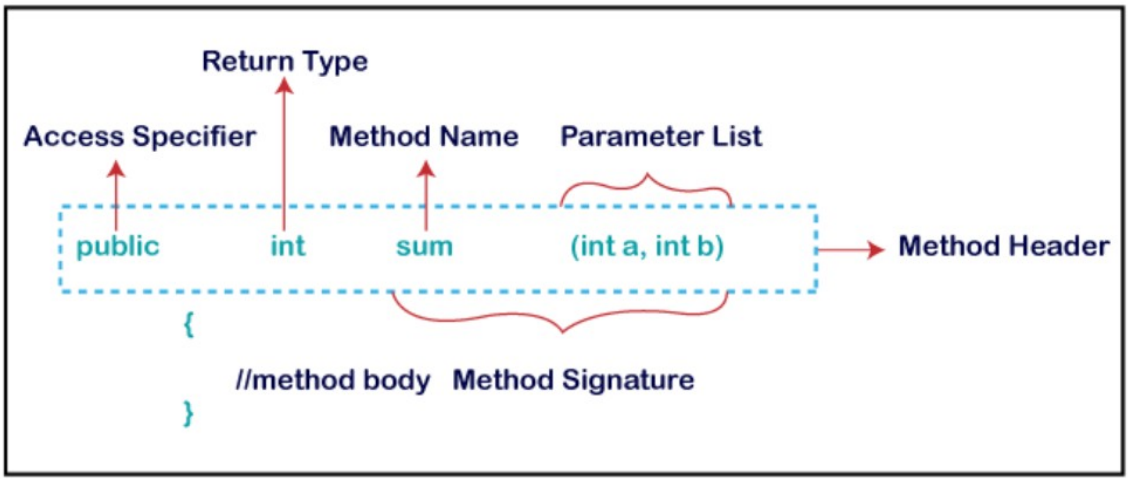
13. Function / Method

a collection of statements that are grouped together to perform an operation.

Advantage of methods **Reusability**

Int Result = Sum(5,7);

Method Signature



Method Call

Method Implementation

The **Method Call** is the method that contains the actual call; the **Method Implementation** is the method that contains code.

1. Access Modifier / Access Specifier

- Public
- Private
- Protected
- Default

2. Class

The collection of **data members / properties / variables / attributes / Fields** and **member functions/ behaviour / methods** is called Class. Sometimes it may also contain **Constructors.**

For example: in real life, a car is an object. The car has attributes, such as Tyres, Seats Color, AC and so on and methods, such as Drive, Brake, Accelerator and so on. A Class is like an object constructor, or a "blueprint" for creating objects.

Syntax:

```
 public class ClassName{
// data members and methods, sometime constructors
}
```

Ex:

```
Public class EmployeeDetails{
// data members and methods, sometime constructors
}
```

3. Object

The instance of Class is called object;

Syntax:

```
ClassName objectname = new ClassName();
```

Ex:

```
EmployeeDetails employeeDetails = new EmployeeDetails();
```

4. Encapsulation

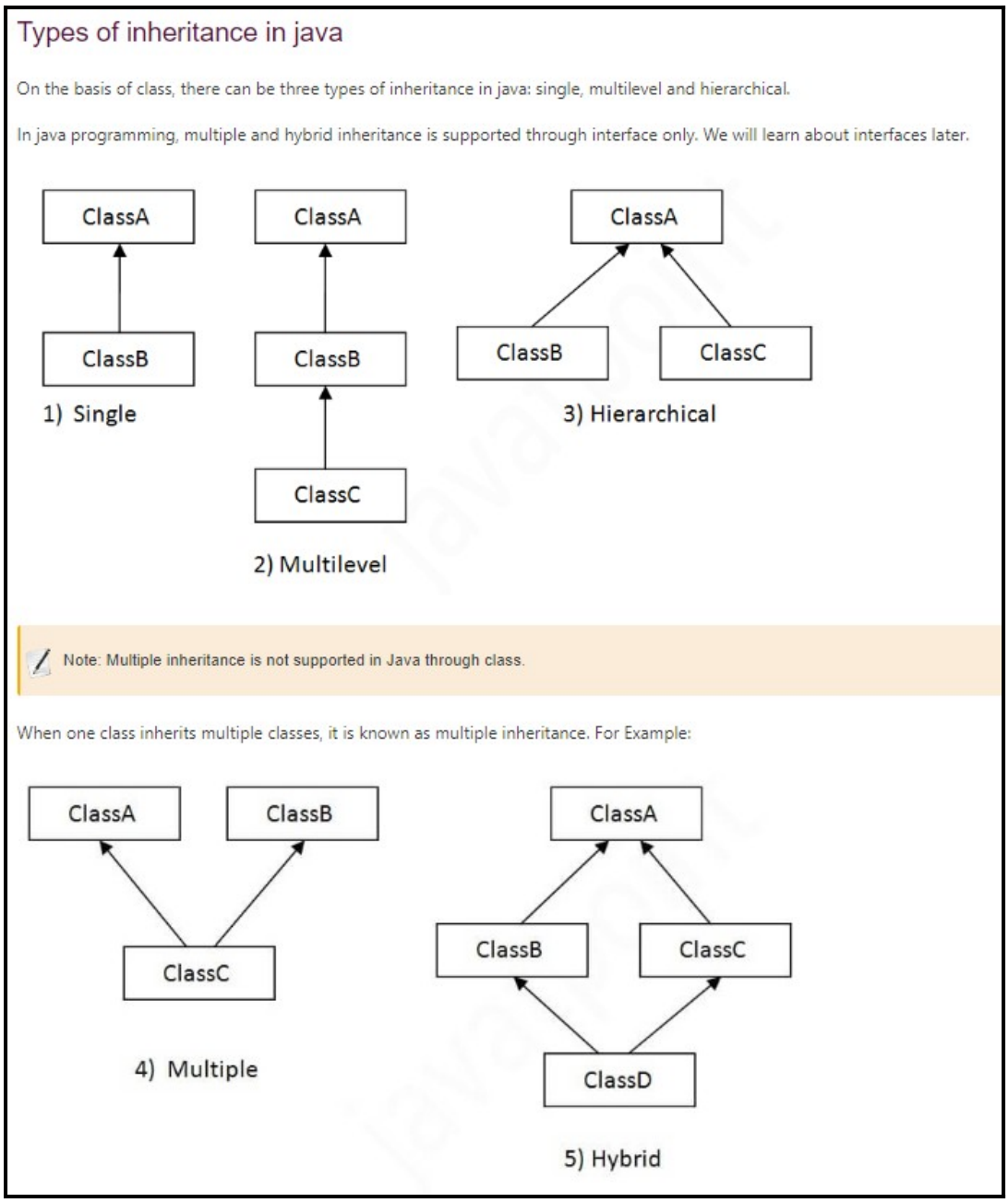
Encapsulation in Java is a process of wrapping code and data together into a single unit **or** binding data with code into a single unit is called encapsulation.

Through encapsulation we are achieving the **data hiding**.

We need to implement getter and setter method to assign and retrieve the values to the data members.

5. Inheritance

Deriving properties and behaviours from one class to another class is called Inheritance. The “extends” keyword is used to inherit.



6. Polymorphism

1. Method Overloading
2. Method Overriding

Method Overloading

Is nothing but same method name but different signature

Signature means – no of arguments, type of arguments and order of arguments.

Method Overriding

Is nothing both method name and signature are same

Car – parent class

Power window,

Power break

Accelatoring.

Maruthi extends car

Accelatoring()

{

Improved facility}

Power window

Power break

Honda

Power break(){

}

7. Abstraction (not complete details)

Hiding the method implementation and **showing only the functionality name** to the user is called abstraction.

It is achieved via

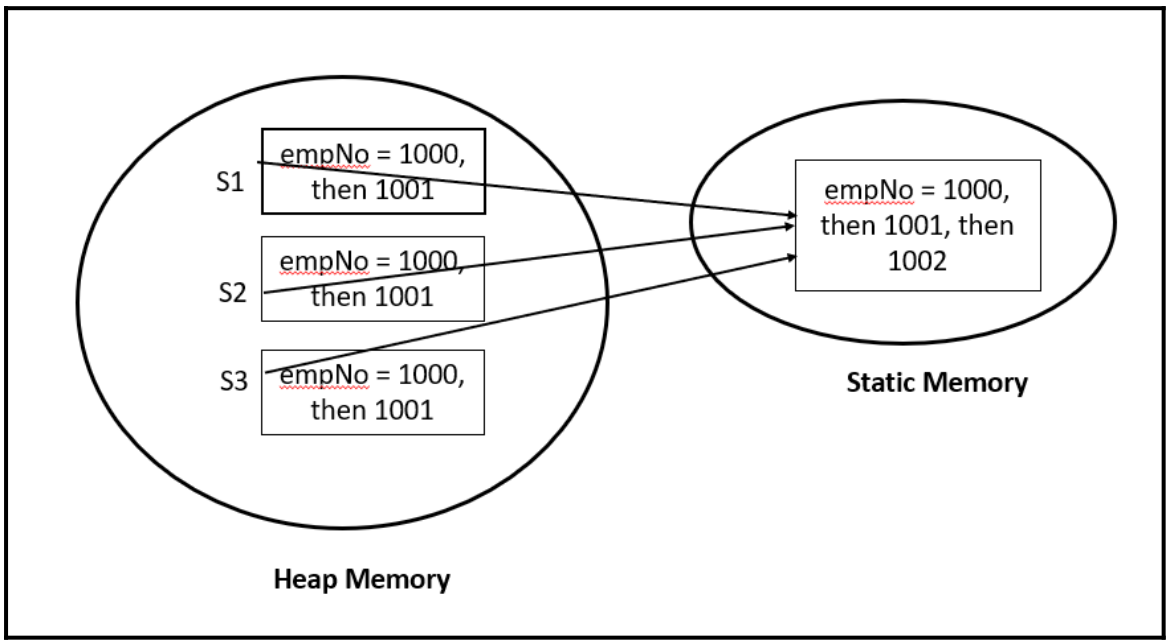
1. Abstract Class – partial abstraction (0 to 100%)
2. Interface - Full abstraction (100%)

Interface

8. Constructor

9. Static

Static variables and objects are stored in static memory which is shared across objects



Final

The variable declared final is constant value and it cannot be changed

10. This

11. Final

14. Collections:

Array

Int[] a= new int[10]; 4*10 = 40 bytes memory allocated

3 * 4 = 12 bytes occupied, remaining 28 bytes of memory unutilized

Collection:

Dynamic memory allocation

Wrapper Class

Int – Integer

float – Float

double – Double

String

char – Character

long - Long

user defined class → EmployeeInfo

ArrayList<Integer> aList = new ArrayList<Integer>();

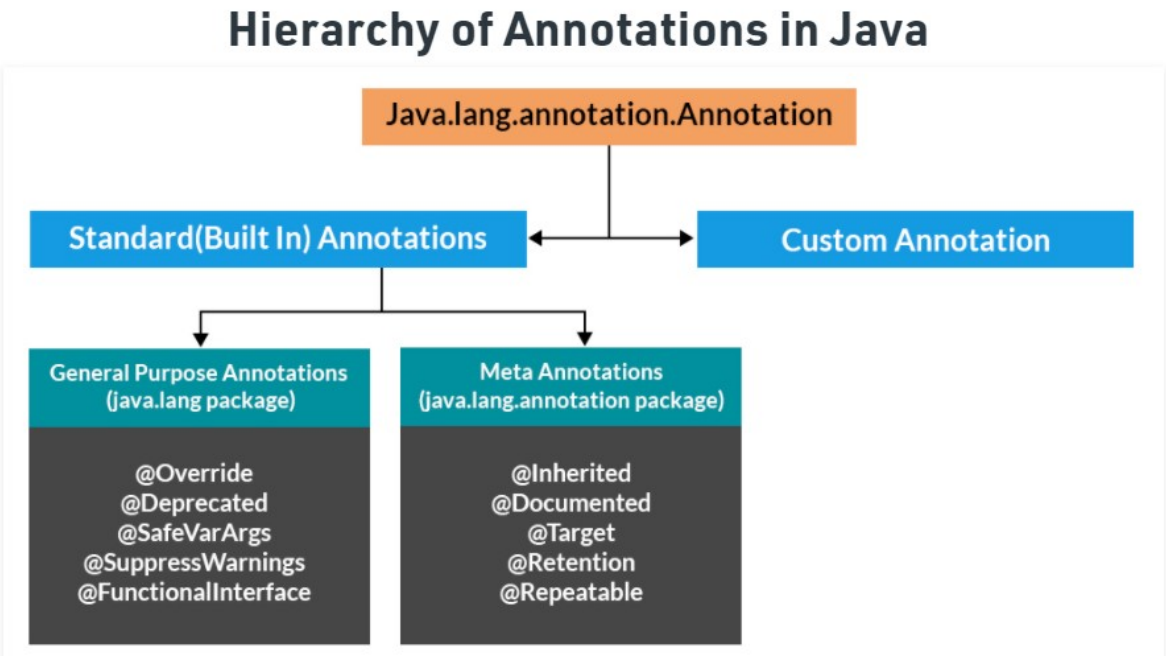
15. Comparable vs Comparator

Both are interfaces;

Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** metho d.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

16. Annotations:

Java **Annotation** is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM. Annotations in Java are used to provide additional information, so it is an alternative option for XML and Java marker interfaces.



17. Multithreading

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

1. Creating and Managing Threads

1. Creating Threads

Threads can be created by using **two mechanisms** :

1. Extending the Thread class
2. Implementing the Runnable Interface

1. Extending the Thread Class

By extending the Thread class, you override the run() method to define the task for the thread.

Example:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread " +
Thread.currentThread().getName() + " is running.");
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread thread1 = new MyThread();
        MyThread thread2 = new MyThread();

        thread1.start(); // Start thread1
        thread2.start(); // Start thread2
    }
}
```

---

2. Implementing the Runnable Interface

By implementing Runnable, you create a task and pass it to a Thread object.

Example:

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread " +
Thread.currentThread().getName() + " is running.");
    }
}

public class Main {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new MyRunnable());
        Thread thread2 = new Thread(new MyRunnable());

        thread1.start(); // Start thread1
        thread2.start(); // Start thread2
    }
}
```

---

2. Managing Threads

A. Starting Threads

- Use the start() method to begin execution of the thread.
- The thread's run() method will execute after start() is called.

B. Joining Threads

- The join() method ensures the current thread waits for another thread to finish.

C. Synchronizing Threads

- When multiple threads access shared resources, synchronization is necessary to avoid race conditions.

D. Thread Priorities

- Threads can have priorities (range: 1 to 10). By default, all threads have priority 5.

Best Practices

1. **Use Synchronization** to avoid race conditions when accessing shared resources.
2. Prefer **Executor Framework** for managing threads instead of manually creating them.
3. Handle **exceptions** in threads to avoid unexpected crashes.
4. Minimize the number of threads to prevent **CPU contention**.

2. `Thread Lifecycle: States and Transitions

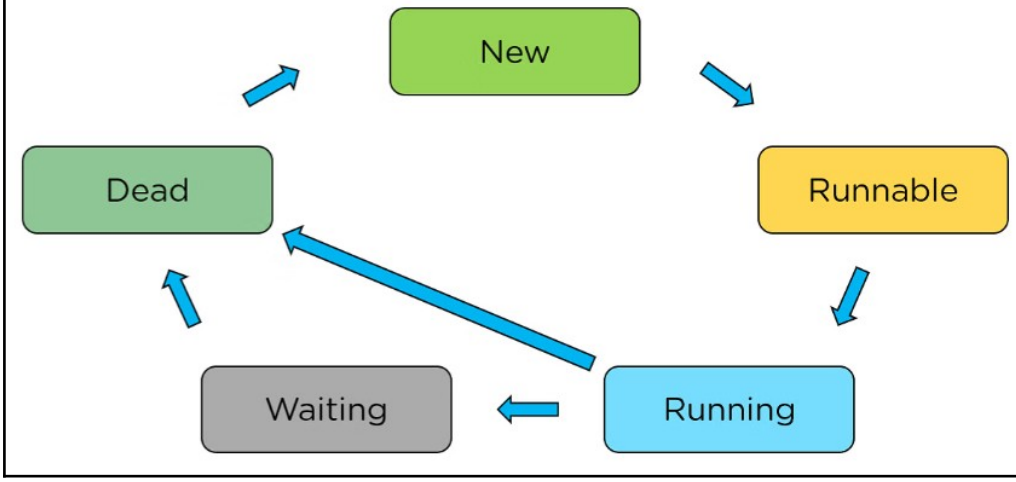
States of a Thread:

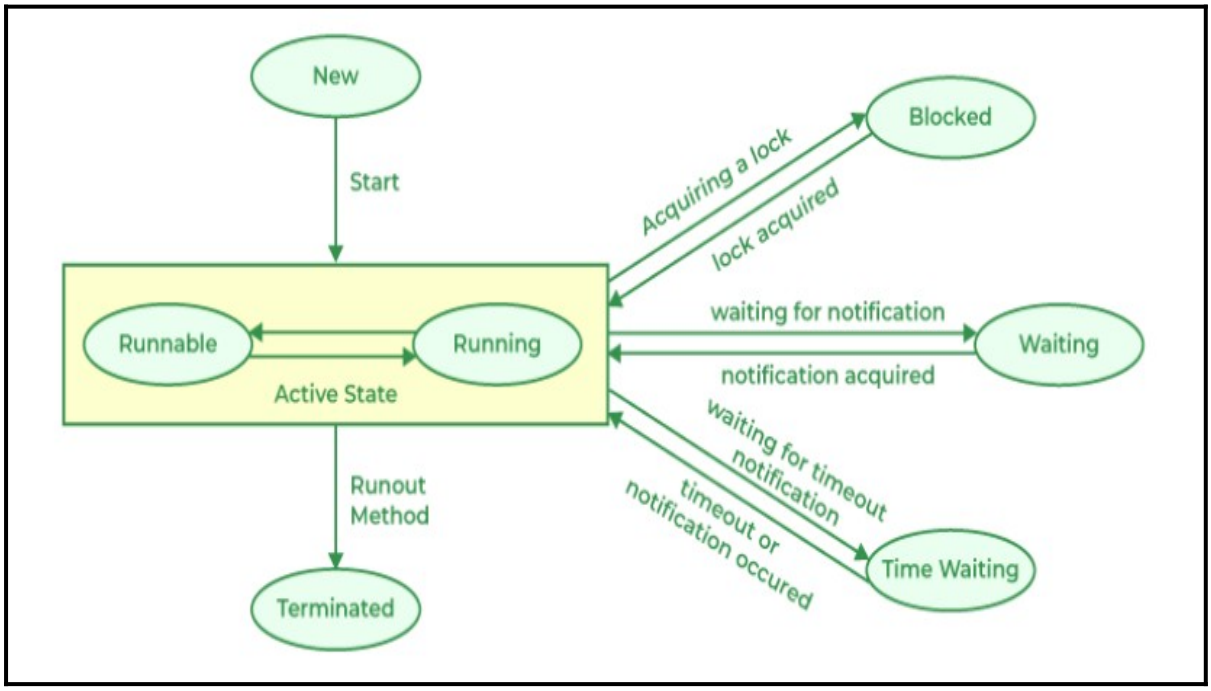
1. **NEW**: Thread is created but not yet started (Thread t = new Thread();).
2. **RUNNABLE**: Thread is ready to run but waiting for CPU time.
3. **RUNNING**: Thread is executing.
4. **BLOCKED/WAITING**: Thread is waiting for a resource or signal.
5. **TERMINATED**: Thread has finished execution.

Lifecycle of a Thread in Java

The lifecycle of each thread in Java has five different stages. You will look into each one of those stages in detail. The Stages of the Lifecycle are mentioned below.

- New
- Runnable
- Running
- Waiting
- Dead





Please Join in below link

Instagram: https://www.instagram.com/rba_infotech/
LinkedIn: https://www.linkedin.com/company/rba-infotech/
Facebook:
https://www.facebook.com/people/RBA-Infotech/100043552406579/

3. Synchronization

1. What is Synchronization

When multiple threads access shared resources, synchronization is necessary to avoid race conditions.

2. Why Synchronization?

When multiple threads access shared resources (e.g., variables, files, or databases), it may lead to **race conditions** or inconsistent data. Synchronization ensures only one thread can access a critical section of code at a time.

3. Real-time Example:

1. Online Movie Ticket Booking System

Consider an online movie ticket booking system where multiple users are trying to book seats for the same show. Here's how synchronization plays a critical role:

1. **Problem Without Synchronization**:
    - Two users (User A and User B) access the system simultaneously to book the last available seat.
    - Both threads (representing the users' actions) check the availability of the seat at the same time and find it available.
    - Both threads proceed to book the seat, leading to **overbooking**, which is an inconsistent state.
2. **Solution With Synchronization**:
    - A **lock** is placed on the seat booking process.
    - When User A starts booking the seat, the lock prevents any other user (User B) from accessing the booking process until User A completes their transaction.
    - After User A books the seat, the system updates the availability and releases the lock.
    - When User B tries to book, the system sees that the seat is no longer available.

Technical Implementation (Example in Java)

Using synchronized to handle this scenario:

```
class TicketBookingSystem {
    private int availableSeats = 1;

    public synchronized void bookSeat(String user) {
        if (availableSeats > 0) {
            System.out.println(user + " is booking a seat...");
            availableSeats--;
            System.out.println(user + " successfully booked a seat. Remaining seats: " + availableSeats);
        } else {
            System.out.println(user + " tried to book a seat, but no seats are available.");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        TicketBookingSystem system = new TicketBookingSystem();

        Thread userA = new Thread(() -> system.bookSeat("User A"));
        Thread userB = new Thread(() -> system.bookSeat("User B"));

        userA.start();
        userB.start();
    }
}
```

2. Multi-threaded Programming (Software):

- **Bank Account Transaction:** Imagine two threads trying to access the same bank account simultaneously. One thread wants to deposit money, while the other wants to withdraw. Without synchronization, the following could happen:
    1. Thread 1 (Withdrawal) reads the balance. 1000 rs
    2. Thread 2 (Deposit) reads the same balance.- 1000 rs
    3. Thread 2 adds the deposit amount to the balance and writes it back. – 1000 + 500 = 1500
    4. Thread 1 subtracts the withdrawal amount from the *original* balance (which is now outdated) and writes it back.- 1000 – 200 = 800

This results in a "lost update" – the deposit is effectively overwritten. Synchronization mechanisms like locks or mutexes ensure that only one thread can access and modify the account balance at any given time, preventing this data corruption.

- **Shared Data Structure:** Consider a list that multiple threads are adding and removing elements from. Without synchronization, you could have race conditions where one thread is iterating through the list while another is modifying it, leading to exceptions or unpredictable behavior. Synchronization ensures that these operations are performed in a thread-safe manner.

2. Operating Systems:

- **Printer Access:** Multiple processes might want to print documents simultaneously. The operating system uses synchronization (e.g., semaphores or mutexes) to manage access to the printer, ensuring that only one process can print at a time and preventing garbled output.
- **File Access:** When multiple processes need to write to the same file, synchronization mechanisms are used to prevent data corruption. For example, file locking can be used to ensure that only one process can write to the file at a time.

3. Databases:

- **Concurrent Transactions:** Databases use synchronization to manage concurrent transactions. If two transactions try to update the same data simultaneously, the database uses locking or other concurrency control mechanisms to ensure data consistency and prevent conflicts.

4. Real-World Analogies:

- **Single-Lane Bridge:** Imagine a narrow bridge that can only accommodate one car at a time. A traffic light or a human flagger acts as a synchronization mechanism, ensuring that cars from opposite directions don't try to cross the bridge simultaneously, preventing a collision.

4. Synchronized Methods

The synchronized keyword can be used to lock a method so only one thread can execute it at a time.

Example:

```
class Counter {
    private int count = 0;

    public synchronized void increment() { // Synchronized method
        count++;
    }

    public int getCount() {
        return count;
    }
}

public class Main {
    public static void main(String[] args) {
        Counter counter = new Counter();
```

```
        Thread thread1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        Thread thread2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        thread1.start();
        thread2.start();

        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Final count: " + counter.getCount());
    }
}
```

---

5. Synchronized Blocks

Synchronized blocks allow you to synchronize only a specific part of a method, rather than the entire method. This can improve performance by reducing the time that threads hold the lock.

Example:

```
class BankAccount {
    private int balance = 1000;

    public void withdraw(String user, int amount) {
        System.out.println(user + " is attempting to withdraw $" +
amount);

        synchronized (this) { // Synchronized block
            if (balance >= amount) {
                System.out.println(user + " is withdrawing $" + amount);
                balance -= amount;
                System.out.println(user + " successfully withdrew $" +
amount + ". Remaining balance: $" + balance);
            } else {
                System.out.println(user + " tried to withdraw $" + amount
+ " but insufficient balance. Remaining balance: $" + balance);
            }
        }

        System.out.println(user + " has completed the transaction.");
    }
}

public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();

        Thread userA = new Thread(() -> account.withdraw("User A",
700));
```

```
        Thread userB = new Thread(() -> account.withdraw("User B",
500));

        userA.start();
        userB.start();
    }
}
```

---

6. Static Synchronization

To synchronize a static method, use the class-level lock.

Example:

```
class Counter {
    private static int count = 0;

    public static synchronized void increment() {
        count++;
    }

    public static int getCount() {
        return count;
    }
}
```

---

7. Deadlock

Deadlock occurs when two or more threads are waiting for each other to release locks, resulting in a standstill.

Example of Deadlock:

```
class DeadlockExample {
    private final Object lock1 = new Object();
    private final Object lock2 = new Object();
```

```
    public void method1() {
        synchronized (lock1) {
            System.out.println("Thread1: Holding lock1...");
            synchronized (lock2) {
                System.out.println("Thread1: Holding lock2...");
            }
        }
    }

    public void method2() {
        synchronized (lock2) {
            System.out.println("Thread2: Holding lock2...");
            synchronized (lock1) {
                System.out.println("Thread2: Holding lock1...");
            }
        }
    }
}
```

---

4. Inter-thread Communication

Inter-thread communication in Java allows threads to communicate with each other, ensuring proper synchronization and coordination. The most common way to achieve this is by using methods like wait(), notify(), and notifyAll() from the Object class.

---

1. Key Points

1. **wait()**: Causes the current thread to release the lock and wait until another thread calls notify() or notifyAll() on the same object.

2. **notify()**: Wakes up a single thread waiting on the object's monitor.
3. **notifyAll()**: Wakes up all threads waiting on the object's monitor.
4. These methods must be called within a synchronized block or method.

---

2. Example: Producer-Consumer Problem

This example demonstrates inter-thread communication where one thread (Producer) produces items, and another thread (Consumer) consumes them.

Code Example

```
class SharedResource {
    private int item = 0; // Shared resource
    private boolean hasItem = false;

    public synchronized void produce() throws InterruptedException {
        while (hasItem) {
            wait(); // Wait until the consumer consumes the item
        }
        item++;
        System.out.println("Produced item: " + item);
        hasItem = true;
        notify(); // Notify the consumer that an item is available
    }

    public synchronized void consume() throws InterruptedException
{
        while (!hasItem) {
            wait(); // Wait until the producer produces an item
```

```
        }
        System.out.println("Consumed item: " + item);
        hasItem = false;
        notify(); // Notify the producer that the item is consumed
    }
}

public class Main {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();

        Thread producer = new Thread(() -> {
            try {
                for (int i = 0; i < 5; i++) {
                    resource.produce();
                    Thread.sleep(500); // Simulate production time
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        Thread consumer = new Thread(() -> {
            try {
                for (int i = 0; i < 5; i++) {
                    resource.consume();
                    Thread.sleep(1000); // Simulate consumption time
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
```

```
            }
        });

        producer.start();
        consumer.start();
    }
}
```

Output Example

```
Produced item: 1
Consumed item: 1
Produced item: 2
Consumed item: 2
Produced item: 3
Consumed item: 3
Produced item: 4
Consumed item: 4
Produced item: 5
Consumed item: 5
```

---

3. How It Works

1. The Producer thread calls the produce() method to produce an item. If an item already exists (hasItem == true), it waits until the Consumer consumes it.
2. The Consumer thread calls the consume() method to consume the item. If no item exists (hasItem == false), it waits until the Producer produces an item.
3. The notify() method in both produce() and consume() ensures that the waiting thread gets notified to proceed.

4. When to use notifyAll()

Multiple Waiting Threads with Different Conditions: When multiple threads are waiting on the same monitor but for *different* conditions to become true, notifyAll() is necessary. If you use notify(), you might wake up a thread that still cannot proceed, leading to inefficiency.

Example (Producer-Consumer with Multiple Consumer Types): Imagine a producer-consumer scenario where you have different types of consumers (e.g., one consumer processes even numbers, another processes odd numbers). When the producer adds a new item, it needs to wake up *all* consumers so that the appropriate consumer can pick up the item.

---

5. Advantages of Inter-thread Communication

- Prevents busy waiting by allowing threads to release the lock and wait efficiently.
- Ensures proper coordination between threads in shared resource scenarios.

This approach is particularly useful in scenarios like job scheduling, message passing, and task queues.

5. Concurrency Utilities

Java provides **Concurrency Utilities** in the java.util.concurrent package to make working with multithreaded and concurrent applications easier, safer, and more efficient. These include tools for managing threads, collections for concurrent use, and advanced asynchronous programming techniques. Below is a breakdown of three important features:

---

1. Executors

The **Executor Framework** provides a higher-level alternative to managing thread pools directly and task execution.

Key Features

- **Thread Pool Management**: Efficiently manages a pool of threads to execute tasks.
- **Task Submission**: Submit tasks using Runnable, Callable, or Future.
- **Ease of Use**: Simplifies thread management by abstracting the low-level details.

Example: Using Executors

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;


public class ExecutorsExample {
    public static void main(String[] args) {
        ExecutorService executor =
Executors.newFixedThreadPool(3); // Thread pool with 3 threads


        for (int i = 1; i <= 5; i++) {
            int taskId = i;
            executor.submit(() -> {
                System.out.println("Task " + taskId + " is being executed
by " + Thread.currentThread().getName());
            });
        }


        executor.shutdown(); // Shutdown the executor
    }
}
```

Output

```
Task 1 is being executed by pool-1-thread-1
Task 2 is being executed by pool-1-thread-2
Task 3 is being executed by pool-1-thread-3
Task 4 is being executed by pool-1-thread-1
```

Task 5 is being executed by pool-1-thread-2

---

2. Concurrent Collections

Standard Java collections (like ArrayList, HashMap) are not thread-safe. Concurrent collections provide thread-safe alternatives that are designed for concurrent access by multiple threads.

Common Classes

1. **ConcurrentHashMap**: A thread-safe hash map that provides high concurrency.
2. **ConcurrentLinkedQueue:** A thread-safe queue.
3. **CopyOnWriteArrayList**: A thread-safe list that creates a new copy of the underlying array whenever a modification is made. Suitable for read-heavy scenarios..
4. **BlockingQueue**: Queues like ArrayBlockingQueue and LinkedBlockingQueue allow safe producer-consumer implementations.

Example: Using ConcurrentHashMap

```
import java.util.concurrent.ConcurrentHashMap;


public class ConcurrentHashMapExample {
    public static void main(String[] args) {
        ConcurrentHashMap<String, Integer> map = new
ConcurrentHashMap<>();


        Thread writerThread = new Thread(() -> {
            map.put("A", 1);
            map.put("B", 2);
            map.put("C", 3);
            System.out.println("Writer added entries.");
```

```
        });

        Thread readerThread = new Thread(() -> {
            map.forEach((key, value) -> System.out.println(key + ": " +
value));
        });

        writerThread.start();
        readerThread.start();
    }
}
```

Output

Writer added entries.

A: 1

B: 2

C: 3

---

3. CompletableFuture

CompletableFuture provides a powerful way to work with asynchronous computations. It allows you to chain asynchronous operations, handle exceptions, and combine results from multiple asynchronous tasks.

Key Features

- Non-blocking: Allows chaining of actions that run asynchronously.
- Supports callbacks: Perform actions upon completion.
- Combine multiple futures: Compose multiple async tasks.

Key Methods:

- **supplyAsync():** Creates a CompletableFuture that executes a supplier asynchronously.
- **thenApply():** Applies a function to the result of the CompletableFuture.

- **thenAccept():** Consumes the result of the CompletableFuture.
- **thenCompose():** Chains two CompletableFuture instances together.
- **exceptionally():** Handles exceptions that occur during the computation.
- **allOf():** Combines multiple CompletableFuture instances into a single CompletableFuture that completes when all of the input instances complete.

Example:

```
import java.util.concurrent.CompletableFuture;
public class CompletableFutureExample {
    public static void main(String[] args) {
        CompletableFuture<Void> future =
CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "Hello";
        }).thenApply(s -> s + " World")
          .thenAccept(System.out::println);
        future.join(); // Wait for completion (no need for try-catch)
        System.out.println("Main thread continues");
    }
}
```

Output:

```
Main thread is free to do other work.
Fetching data from a remote API...
Processing the data: Data from API
```

Final result: DATA FROM API

Summary Table

Feature	Use Case	Example Class/Method
Executors	Thread management and task execution	ExecutorService, submit
Concurrent Collections	Safe data structures for concurrent access	ConcurrentHashMap, BlockingQueue
CompletableFuture	Asynchronous programming with non-blocking operations	supplyAsync, thenApply

These utilities help write clean, scalable, and efficient concurrent applications, making Java a powerful tool for modern, multi-threaded programming.

6. Thread Pools

Thread pools are a crucial concept in concurrent programming, especially in Java. They provide a way to manage and reuse threads efficiently, significantly improving performance and resource utilization compared to creating new threads for every task.

1. Why Use Thread Pools?

Creating a new thread for each task can be expensive in terms of system resources (CPU time, memory). Thread creation and destruction involve operating system overhead. If you have a large number of short-lived tasks, the overhead of managing threads can outweigh the benefits of concurrency.

Thread pools address this issue by:

- **Reusing Threads:** Instead of creating a new thread for each task, a pool of pre-created threads is maintained. When a task arrives, a thread from the pool is assigned to it. Once the task is completed, the thread is returned to the pool, ready to be used for another task.

- **Limiting the Number of Threads:** Thread pools allow you to control the maximum number of threads that can be created. This prevents the system from being overwhelmed by too many threads, which can lead to performance degradation or even system crashes.
- **Managing Thread Lifecycle:** The thread pool manages the lifecycle of the threads, including creation, execution, and termination. This simplifies the development of concurrent applications.

2. Key Components of a Thread Pool:

- **Thread Pool:** A collection of worker threads.
- **Task Queue:** A queue that holds tasks waiting to be executed.
- **Thread Factory (Optional):** An object that creates new threads for the pool.
- **Rejection Policy (Optional):** A strategy for handling tasks that cannot be accepted by the pool (e.g., when the queue is full).

3. How Thread Pools Work:

1. A task is submitted to the thread pool.
2. If there is an idle thread in the pool, the task is assigned to that thread for execution.
3. If there are no idle threads and the pool has not reached its maximum size, a new thread is created and assigned to the task.
4. If all threads are busy and the pool has reached its maximum size, the task is placed in the task queue.
5. If the queue is also full, the rejection policy is applied.
6. When a thread finishes executing a task, it retrieves the next task from the queue (if any) or becomes idle and waits for new tasks.

4. Types of Thread Pools in Java (using Executors factory class):

- **Executors.newFixedThreadPool(int nThreads):** Creates a thread pool with a fixed number of threads. Tasks are submitted to a queue. If all threads are busy, new tasks will wait in the queue until a thread becomes available.
- **Executors.newCachedThreadPool():** Creates a thread pool that creates new threads as needed, but reuses previously created threads when they are available. If a thread is idle for a certain period, it is removed from the pool. This is suitable for short-lived tasks.
- **Executors.newSingleThreadExecutor():** Creates a thread pool with a single thread. This ensures that tasks are executed sequentially in the order they are submitted.
- **Executors.newScheduledThreadPool(int corePoolSize):** Creates a thread pool that can schedule tasks to run after a certain delay or periodically.

5. Important Considerations:

- **Shutdown the Executor:** It's crucial to shut down the executor when you're finished with it using executor.shutdown() or executor.shutdownNow(). This prevents the application from hanging because the threads in the pool might keep running.
- **Choosing the Right Pool Type:** The best type of thread pool depends on the characteristics of your tasks. Consider factors like the number of tasks, the duration of tasks, and the available system resources.
- **Tuning Thread Pool Parameters:** For more advanced scenarios, you can use ThreadPoolExecutor directly to fine-tune parameters like core pool size, maximum pool size, keep-alive time, and queue capacity.

Thread pools are a fundamental tool for writing efficient concurrent applications in Java. They provide a structured and controlled way to manage threads, leading to improved performance and resource utilization.

7. Thread Safety

1. Wring Thread Safe Code

Thread safety ensures that a piece of code behaves correctly when accessed by multiple threads simultaneously. Writing thread-safe code requires strategies to prevent race conditions, data corruption, or unexpected behaviour. One key approach is using **immutable objects**, which are inherently thread-safe.

---

1. What is Thread Safety?

A piece of code or object is **thread-safe** if it functions correctly when accessed by multiple threads, regardless of the timing or interleaving of those threads.

Challenges in Thread Safety

1. **Race Conditions**: Occur when two or more threads access shared data simultaneously, and at least one modifies it.
2. **Deadlocks**: Occur when threads wait indefinitely for locks held by each other.
3. **Visibility Issues**: Changes made by one thread may not be visible to others due to caching or reordering by the JVM.

---

2. Strategies for Writing Thread-Safe Code

A. Synchronization

Use the synchronized keyword or explicit locks to ensure only one thread can execute a critical section at a time.

B. Using Volatile Variables

The volatile keyword ensures visibility of changes to a variable across threads. It is suitable for variables where atomicity isn't required.

Example:

public class VolatileExample {

```
    private volatile boolean running = true; // Volatile boolean flag
    public void start() {
        new Thread(() -> {
            while (running) {
                // Do some work
                System.out.println("Worker thread is running...");
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("Worker thread stopped.");
        }).start();
    }
    public void stop() {
        running = false; // Setting volatile variable from another thread
    }
    public static void main(String[] args) throws InterruptedException
{
        VolatileExample example = new VolatileExample();
        example.start();
        Thread.sleep(3000);
        System.out.println("Stopping worker thread...");
        example.stop();
    }
}
```

C. Using Atomic Classes

Classes in the java.util.concurrent.atomic package (e.g., AtomicInteger, AtomicReference) provide thread-safe operations without explicit synchronization.

Example:

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger count = new AtomicInteger();

    public void increment() {
        count.incrementAndGet();
    }

    public int getCount() {
        return count.get();
    }
}
```

D. Thread-Safe Collections

Use concurrent collections like ConcurrentHashMap, CopyOnWriteArrayList, or BlockingQueue.

E. Avoiding Shared Mutability

Reduce shared mutable state. Instead, use **immutable objects** or **thread-local variables**.

8. Immutable Object

Immutable objects are inherently thread-safe as their state cannot be modified after creation. Any attempt to change the state results in a new object being created.

1. Characteristics of Immutable Objects

1. All fields are final.
2. The class itself is declared as final.

3. No setters or methods that modify the object.
4. Any mutable fields are safely copied during object creation.

Example:

```
public final class ImmutablePerson { // Make the class final
    private final String name; // Make fields final
    private final int age;
    public ImmutablePerson(String name, int age) { // Constructor
        this.name = name;
        this.age = age;
    }

    public String getName() { // Only getter methods
        return name;
    }
    public int getAge() {
        return age;
    }
    // No setter methods
    public ImmutablePerson withAge(int newAge) {
        return new ImmutablePerson(this.name, newAge);
    }
    public ImmutablePerson withName(String newName) {
        return new ImmutablePerson(newName, this.age);
    }

    public static void main(String[] args) {
        ImmutablePerson person1 = new ImmutablePerson("Alice",
30);
        ImmutablePerson person2 = person1.withAge(31); // Creates a
new object
```

```
        System.out.println(person1.getAge()); // Output: 30
        System.out.println(person2.getAge()); // Output: 31
    }
}
```

---

2. Benefits of Immutable Objects

1. Thread-safe by default.
2. Easier to reason about code as state does not change.
3. Suitable for use as keys in maps or elements in sets.

---

3. Thread Safety Best Practices

A. Minimize Shared State

- Use local variables or thread-local variables (ThreadLocal) instead of shared fields.

Example: Using ThreadLocal

```
class ThreadLocalExample {
    private static ThreadLocal<Integer> threadLocal =
ThreadLocal.withInitial(() -> 0);

    public void increment() {
        threadLocal.set(threadLocal.get() + 1);
    }

    public int getValue() {
        return threadLocal.get();
    }
}
```

B. Use High-Level Concurrency APIs

- Prefer ExecutorService and CompletableFuture over manually managing threads.
- Use java.util.concurrent utilities for synchronization and collections.

C. Prefer Immutability

- Design classes to be immutable when possible.
- Use immutable libraries (e.g., Guava's immutable collections).

D. Avoid Deadlocks

- Use a consistent locking order for multiple locks.
- Use tryLock() with a timeout instead of blocking indefinitely.

E. Test for Thread Safety

- Use tools like ThreadSafe annotations or test libraries to detect concurrency issues.
- Simulate multithreaded access during testing.

Summary

1. **Synchronization** and **atomic classes** help manage shared mutable state.
2. **Immutable objects** eliminate the need for synchronization and make code simpler and thread-safe by design.

Use **concurrent utilities** and modern Java features to avoid low-level thread management.

18. Generics and Type Safety

1. Generic Classes and Methods

Generics in Java

Generics in Java allow you to write classes, interfaces, and methods that can operate on any type while ensuring type safety at compile time. They are part of Java's type system and help reduce runtime errors caused by improper type casting.

---

1. Generic Classes

A **generic class** is a class that defines a type parameter <T> to allow the use of various data types without needing to write separate classes.

Syntax

```
public class GenericClass<T> {
    private T value;

    // Constructor
    public GenericClass(T value) {
        this.value = value;
    }

    // Getter
    public T getValue() {
        return value;
    }

    // Setter
    public void setValue(T value) {
        this.value = value;
    }
}
```

Example Usage

```
public class Main {
    public static void main(String[] args) {
```

```
        // Integer type
        GenericClass<Integer> intObject = new GenericClass<>(42);
        System.out.println("Integer Value: " + intObject.getValue());

        // String type
        GenericClass<String> stringObject = new
GenericClass<>("Hello Generics");
        System.out.println("String Value: " + stringObject.getValue());
    }
}
```

---

2. Generic Methods

A **generic method** is a method that declares a type parameter <T> and works independently of any generic class.

Syntax

```
public class Utility {
    // Generic Method
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

Example Usage

```
public class Main {
    public static void main(String[] args) {
        Integer[] intArray = {1, 2, 3};
        String[] stringArray = {"A", "B", "C"};
```

```
        // Calling the generic method
        Utility.printArray(intArray);
        Utility.printArray(stringArray);
    }
}
```

---

4. Bounded Type Parameters and Wildcards

3. Bounded Generics

Generics can be constrained to work with specific types using the extends keyword.

Syntax

```
public class BoundedGeneric<T extends Number> {
    private T value;

    public BoundedGeneric(T value) {
        this.value = value;
    }

    public double getDoubleValue() {
        return value.doubleValue();
    }
}
```

Example Usage

```
public class Main {
    public static void main(String[] args) {
        BoundedGeneric<Integer> intObject = new
BoundedGeneric<>(42);
        System.out.println("Double Value: " +
intObject.getDoubleValue());
```

```
        // BoundedGeneric<String> stringObject = new
BoundedGeneric<>("Hello"); // Compile-time error
    }
}
```

Wildcards (in Java):

Wildcards (?) are used to represent unknown types in generic type parameters. They provide more flexibility when working with collections of different types.

- ?: Represents any type.
- ? extends Type: Represents any type that is a subtype of Type. (Upper Bounded Wildcard)
- ? super Type: Represents any type that is a supertype of Type. (Lower Bounded Wildcard)

Example (Java with Wildcards):

```
import java.util.List;
import java.util.ArrayList;

public class WildcardExample {
    public static void printList(List<?> list) { // Accepts a list of any
type
        for (Object item : list) {
            System.out.print(item + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        List<Integer> intList = new ArrayList<>();
        intList.add(1);
```

```
        intList.add(2);

        List<String> stringList = new ArrayList<>();
        stringList.add("a");
        stringList.add("b");

        printList(intList);
        printList(stringList);
    }
}
```

4. Wildcards in Generics

Wildcards are used to represent an unknown type and are useful in scenarios where the exact type is not essential.

Types of Wildcards

1. **Unbounded Wildcard**: <?>
    - Accepts any type.
2. public static void printList(List<?> list) {
3. for (Object obj : list) {
4. System.out.println(obj);
5. }
6. }
7. **Bounded Wildcard with Upper Bound**: <? extends Type>
    - Accepts Type or its subclasses.
8. public static double sum(List<? extends Number> numbers) {
9. double total = 0;
10. for (Number num : numbers) {
11. total += num.doubleValue();
12. }
13. return total;

14. }
15. **Bounded Wildcard with Lower Bound**: <? super Type>
    - Accepts Type or its superclasses.
16. public static void addIntegers(List<? super Integer> list) {
17. list.add(10);
18. list.add(20);
19. }

---

3. Key Differences: Bounded Type Parameters vs Wildcards

Aspect	Bounded Type Parameters	Wildcards
Definition	Restricts type parameters during declaration.	Allows flexibility for unknown types during usage.
Syntax	<T extends Type>	<? extends Type> or <? super Type>
Usage	Used in class/method declarations.	Used in method parameters.
Examples	<T extends Number>	List<? extends Number>

Best Practices

1. **Use Bounded Type Parameters**:
    - When defining a class or method and requiring stricter type control.
    - Example: <T extends Comparable<T>>.
2. **Use Wildcards**:
    - When defining methods that operate on unknown or partially known types.

- Example: List<? extends Number> for read-only operations.

3. **Prefer Upper-Bounds for Read-Only**:
   - Use <? extends Type> when you only need to access elements.
4. **Prefer Lower-Bounds for Write-Only**:
   - Use <? super Type> when you only need to modify elements.

---

5. Generic Interfaces

Interfaces can also be generic.

Syntax

```
public interface GenericInterface<T> {
    void display(T value);
}


public class GenericClassImpl implements GenericInterface<String>
{
    public void display(String value) {
        System.out.println("Value: " + value);
    }
}
```

Example Usage

```
public class Main {
    public static void main(String[] args) {
        GenericClassImpl obj = new GenericClassImpl();
        obj.display("Hello Interface Generics");
    }
}
```

---

6. Practical Applications of Generics

1. **Collections Framework**:
    - Generics power the Java Collections Framework classes, such as List, Set, Map.
2. List<String> names = new ArrayList<>();
3. names.add("Alice");
4. names.add("Bob");
5. **Utility Classes**:
    - Writing reusable utility methods or classes for sorting, searching, and more.
6. **Type Safety**:
    - Prevents runtime ClassCastException.

5. Generic Collections and Type Inference in Java

Generics and type inference play a crucial role in Java's **Collections Framework**. They ensure type safety and simplify code by automatically determining the type parameters for generic types during instantiation or method calls.

---

1. Generic Collections

Java's **Collections Framework** supports generics, allowing you to create collections that are type-safe. This eliminates the need for explicit casting and reduces runtime errors.

Examples of Generic Collections

List

```
import java.util.List;
import java.util.ArrayList;


public class Main {
    public static void main(String[] args) {
        List<String> stringList = new ArrayList<>();
        stringList.add("Apple");
```

```
        stringList.add("Banana");

        // Compile-time type safety
        for (String fruit : stringList) {
            System.out.println(fruit);
        }

        // stringList.add(10); // Compile-time error
    }
}
```

Map

```
import java.util.Map;
import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "One");
        map.put(2, "Two");

        for (Map.Entry<Integer, String> entry : map.entrySet()) {
            System.out.println(entry.getKey() + " => " +
entry.getValue());
        }
    }
}
```

Set

```
import java.util.Set;
import java.util.HashSet;
```

```
public class Main {
    public static void main(String[] args) {
        Set<Double> numberSet = new HashSet<>();
        numberSet.add(1.1);
        numberSet.add(2.2);

        for (Double number : numberSet) {
            System.out.println(number);
        }
    }
}
```

---

2. Type Inference

Type Inference allows the compiler to deduce the type parameters of a generic class or method, reducing boilerplate code and improving readability.

Before Java 7

List<String> list = new ArrayList<String>();

From Java 7 (Diamond Operator)

List<String> list = new ArrayList<>();

The **diamond operator (<>)** allows the compiler to infer the generic type from the left-hand side.

Type Inference in Methods

Type inference also applies to methods, where the compiler can determine the generic type based on the method arguments or context.

Example

import java.util.Arrays;

import java.util.List;

```
public class Main {
    public static <T> void printList(List<T> list) {
        for (T item : list) {
            System.out.println(item);
        }
    }

    public static void main(String[] args) {
        List<Integer> intList = Arrays.asList(1, 2, 3);
        List<String> strList = Arrays.asList("A", "B", "C");

        // Type inference in method calls
        printList(intList);
        printList(strList);
    }
}
```

---

3. Generic Collections vs Raw Collections

Raw Collections

Raw collections do not specify a type parameter and are not type-safe. They rely on Object as the default type, leading to potential runtime errors.

Example of Raw Collection

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList rawList = new ArrayList();
        rawList.add("Hello");
        rawList.add(10); // No compile-time error
```

```
        for (Object obj : rawList) {
            System.out.println(obj); // May cause ClassCastException at runtime
        }
    }
}
```

Why Use Generic Collections Instead?

1. **Type Safety**: Prevents adding incorrect types.
2. **No Casting**: Eliminates the need for explicit casting when retrieving elements.
3. **Readability**: Code is easier to understand and maintain.

---

4. Practical Applications

Type-Safe Collections

```
List<Integer> numbers = new ArrayList<>();
numbers.add(1);
numbers.add(2);
// numbers.add("Three"); // Compile-time error
```

Custom Generic Classes with Collections

```
import java.util.List;

public class DataContainer<T> {
    private List<T> items;

    public DataContainer(List<T> items) {
        this.items = items;
    }

    public void displayItems() {
```

```
        for (T item : items) {
            System.out.println(item);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        List<String> stringList = List.of("One", "Two", "Three");
        DataContainer<String> container = new
DataContainer<>(stringList);
        container.displayItems();
    }
}
```

---

5. Advanced Type Inference with Streams

Type inference is extensively used in functional programming, especially with streams.

Example

```
import java.util.List;
import java.util.stream.Collectors;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5);

        // Type inference in streams
        List<String> result = numbers.stream()
                                     .map(n -> "Number: " + n)
                                     .collect(Collectors.toList());
```

```
        result.forEach(System.out::println);
    }
}
```

---

6. Limitations of Type Inference

1. **Ambiguity**: Type inference might fail in complex scenarios.
2. **Requires Explicit Types**: In some cases, you may still need to specify types manually.

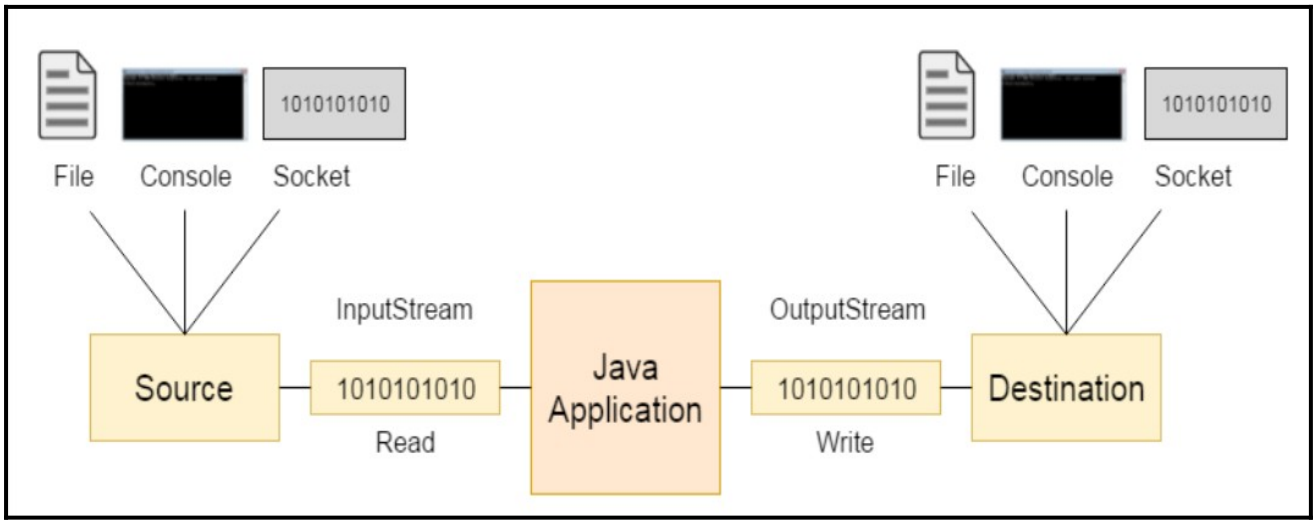
Example of Ambiguity

```
// Compiler cannot infer the type parameter
List<?> list = new ArrayList<>(); // Requires explicit type for clarity
```

Would you like more details on generic collections or further examples of type inference in Java?

19. Java IO Stream

Java IO (Input/Output) is a core part of Java used for reading and writing data. It provides classes and methods for handling different types of input and output operations, including working with files, standard input/output streams, and network sockets.

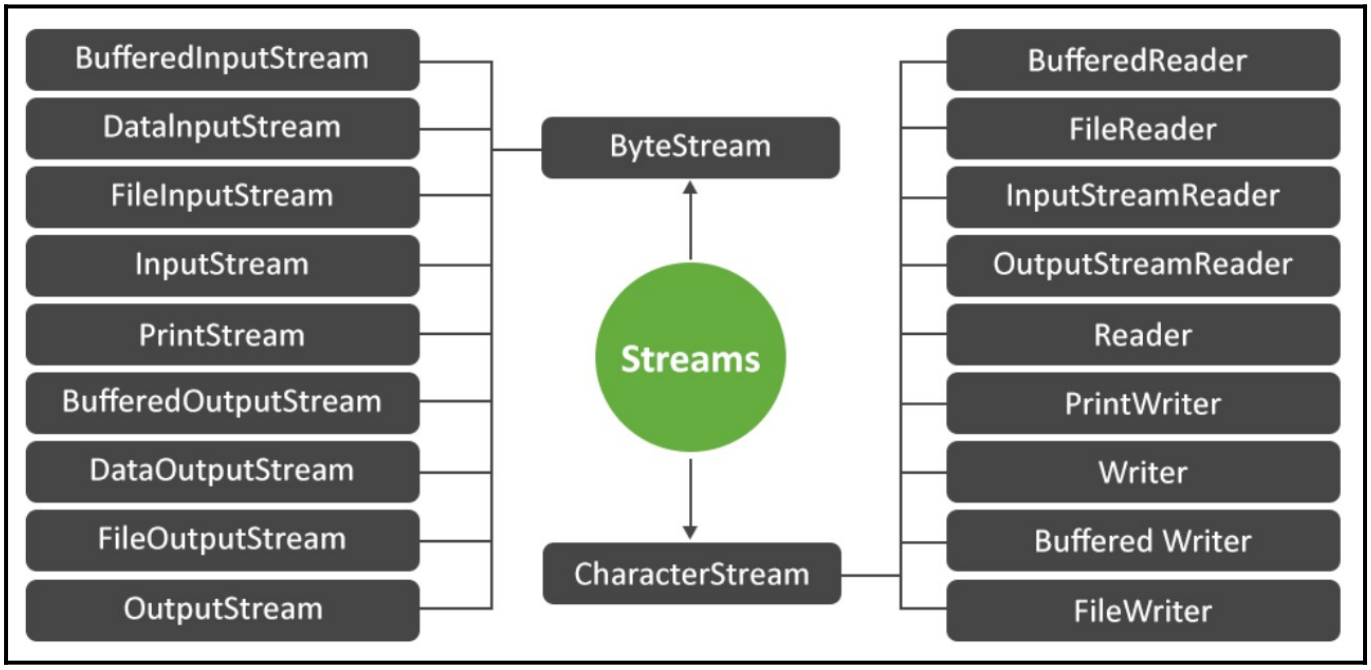


1. Streams

A stream represents a sequence of data. Data can be byte or character. In Java I/O, there are two main types of streams:

Two Types of Streams

- **Byte Streams:** Handle data in the form of bytes (8-bit units). They are used for reading and writing binary data, such as images, audio, and video files.
- **Character Streams:** Handle data in the form of characters (16-bit Unicode units). They are used for reading and writing text data.

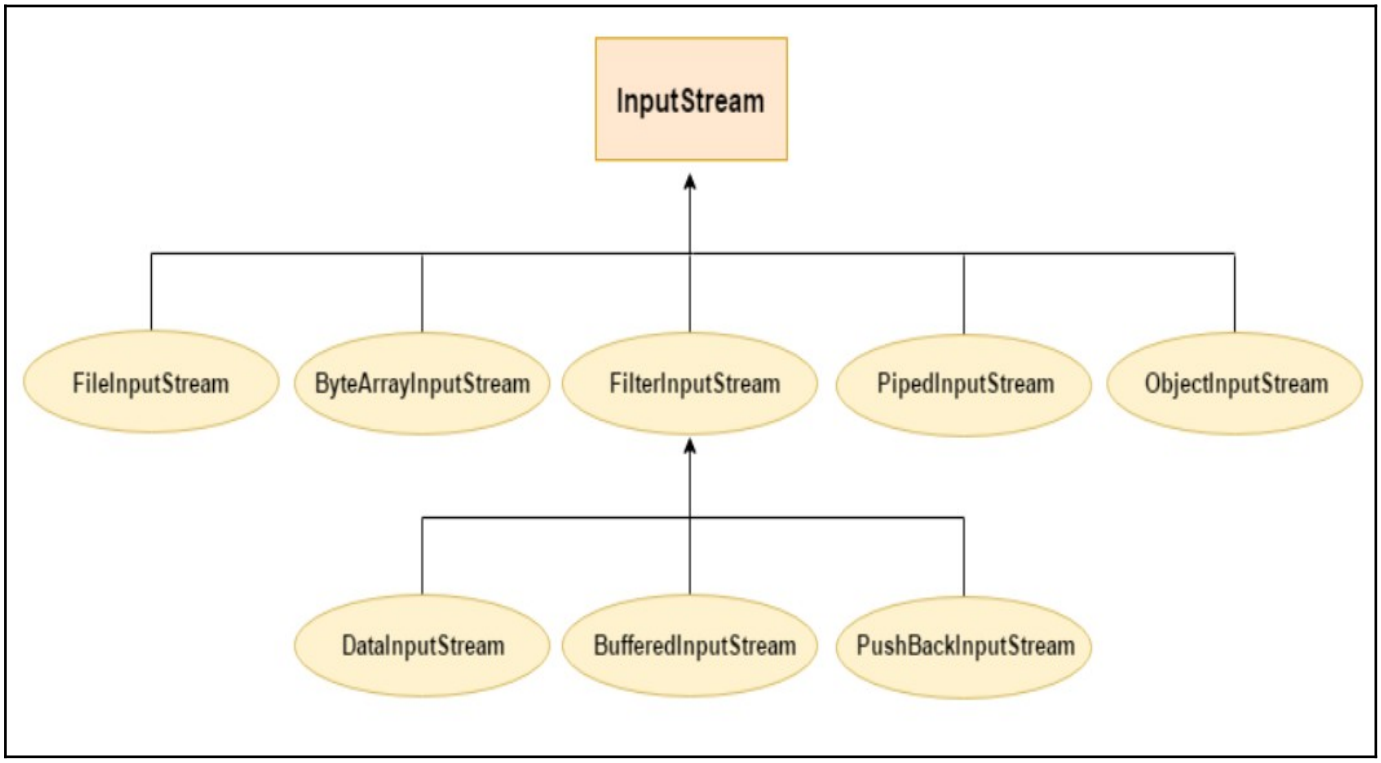


2. Byte Stream

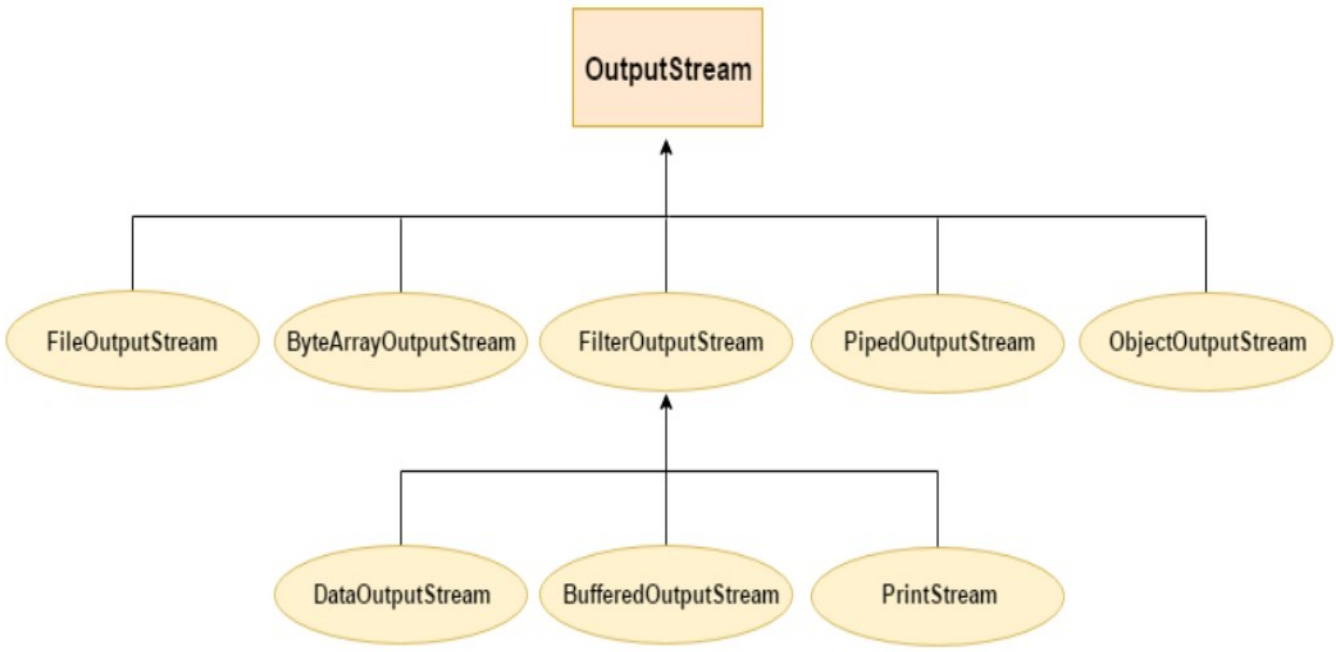
Two subclasses are

1. InputStream
2. OutputStream

1. InputStream



2. OutputStream



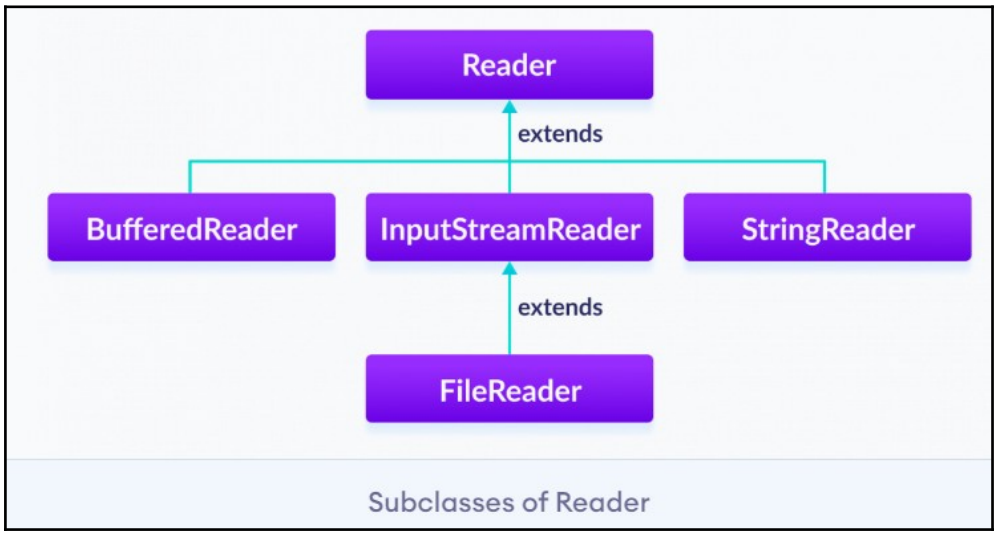
3. Character Steam

It is used to read or write a single character of data

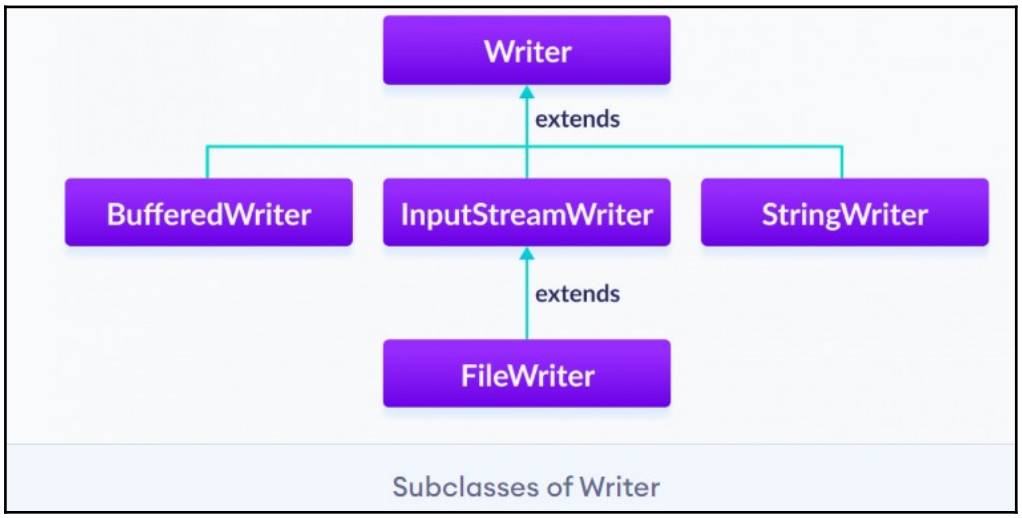
Two type of character streams

1. Reader
2. Writer

3. Reader



4. Writer



4. When to Use Byte Streams:

Use byte streams when you are working with:

- **Binary data:** This includes any data that is not primarily text, such as:
    - Image files (JPEG, PNG, GIF, etc.)
    - Audio files (MP3, WAV, etc.)
    - Video files (MP4, AVI, etc.)
    - Executable files (.exe, .dll, .jar etc.)
    - Compressed files (ZIP, RAR, etc.)
    - Any other file format that stores data in a non-textual format.
- **Data that needs to be transferred exactly as is:** If you need to preserve the exact byte representation of the data (without any character encoding conversions), use byte streams.
- **Network communication:** Network communication often involves sending and receiving raw bytes.

Examples of Byte Stream Classes:

- FileInputStream / FileOutputStream (for file I/O)
- ByteArrayInputStream / ByteArrayOutputStream (for in-memory byte arrays)
- BufferedInputStream / BufferedOutputStream (for buffered byte I/O)

5. When to Use Character Streams:

Use character streams when you are working with:

- **Text data:** This includes:
    - Text files (.txt, .csv, .html, .xml, etc.)
    - Strings
    - Data that represents human-readable text.
- **Data that needs character encoding handling:** Character streams handle character encoding conversions automatically,

ensuring that text is read and written correctly regardless of the underlying byte representation.

Examples of Character Stream Classes:

- FileReader / FileWriter (for file I/O)
- CharArrayReader / CharArrayWriter (for in-memory character arrays)
- BufferedReader / BufferedWriter (for buffered character I/O)
- InputStreamReader / OutputStreamWriter (for bridging between byte streams and character streams, allowing you to specify a character encoding)

Key Differences Summarized:

Feature	Byte Streams	Character Streams
Data Unit	Bytes (8 bits)	Characters (16 bits, Unicode)
Purpose	Binary data, raw data transfer	Text data, character encoding handling
Encoding	No encoding conversion	Handles character encodings (e.g., UTF-8, UTF-16)
Classes	InputStream, OutputStream and subclasses	Reader, Writer and subclasses
Example Use Cases	Image processing, network communication, file copying	Reading/writing text files, string manipulation

Export to Sheets

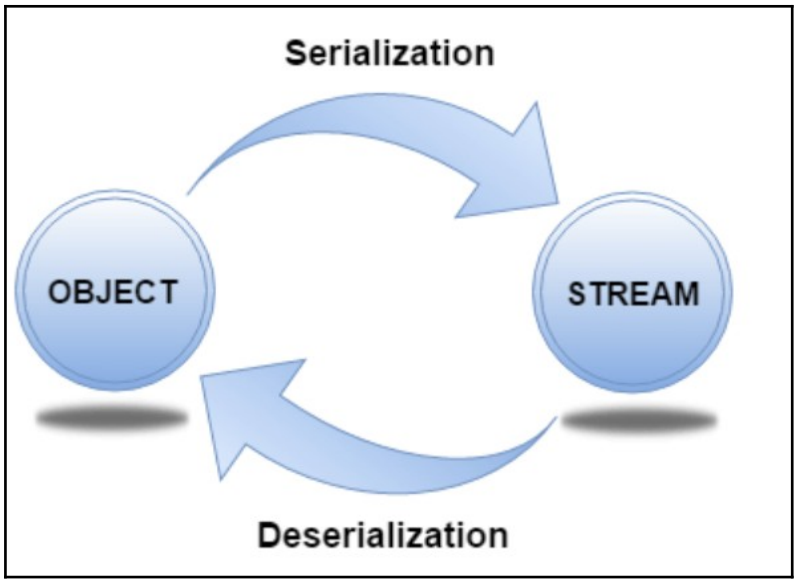
Example Scenario:

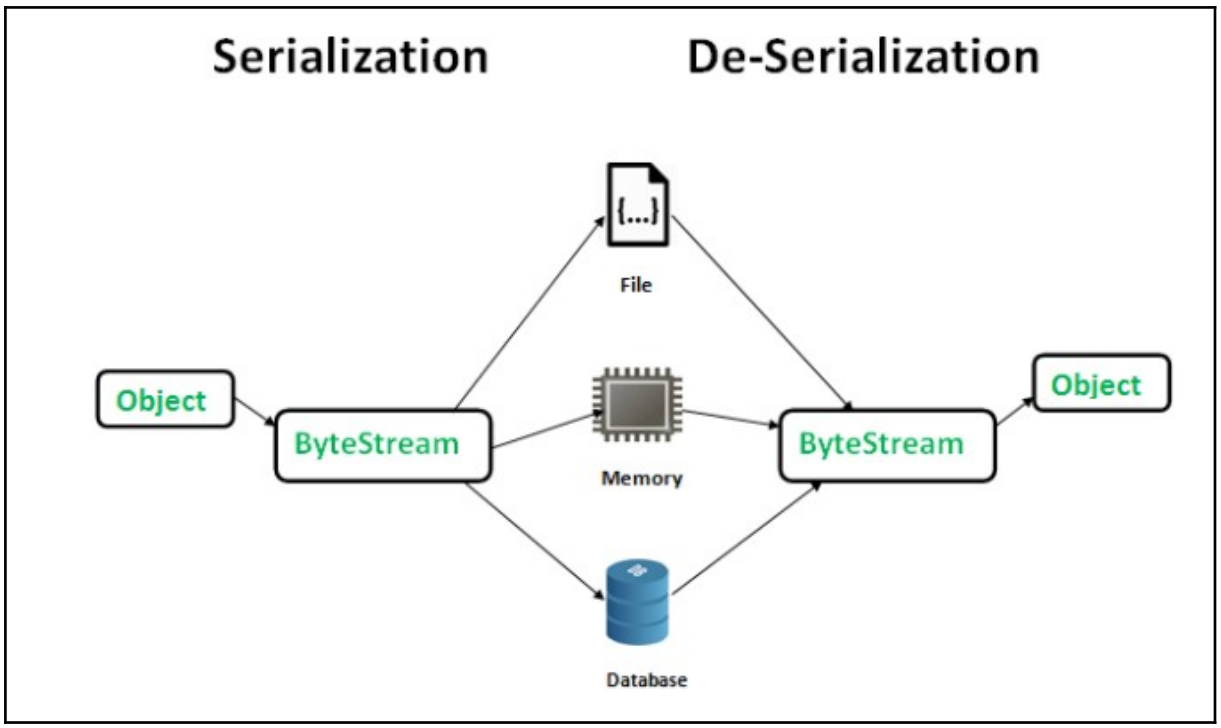
If you are copying an image file, you *must* use byte streams. If you use character streams, the image data will likely be corrupted because character streams will attempt to interpret the byte data as characters, leading to encoding issues.

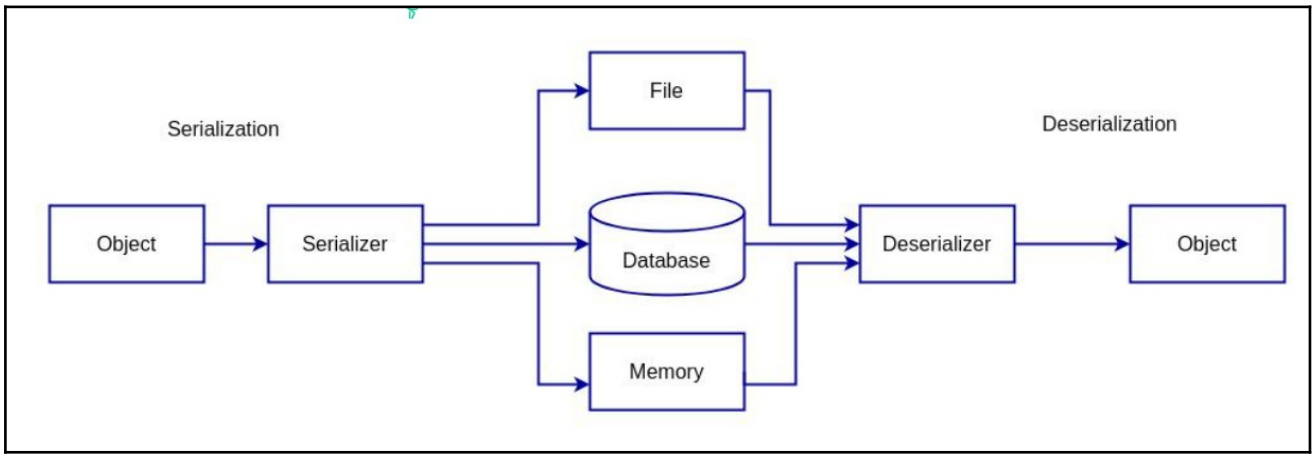
If you are reading a text file that contains characters outside the basic ASCII range (e.g., accented characters, characters from other languages), you *should* use character streams to ensure that the characters are read correctly. If you use byte streams, you would have to handle the character encoding yourself.

In most cases, when dealing with text, character streams are the better choice due to their automatic encoding handling. Byte streams are essential for all other types of data.

6. Serialization and Deserialization







Serialization and **Deserialization** are mechanisms in Java used to convert objects into a format that can be easily stored or transmitted and later reconstructed. These concepts are widely used in scenarios like saving application state, sending objects over a network, or persisting data to a file.

---

1. Serialization

Serialization is the process of converting a Java object into a byte stream. This byte stream can then be:

- **Stored in a file:** For persistence, so the object can be retrieved later.
- **Transmitted over a network:** To send the object to another system.
- **Stored in memory:** For caching or other purposes.
- **Database:** can be stored in database to be retrieved later.

2. Deserialization

Deserialization is the reverse process of serialization. It's the process of reconstructing an object from a byte stream that was previously created through serialization.

How Serialization Works in Java:

1. **Serializable Interface:** To make a Java object serializable, its class must implement the java.io.Serializable interface. This is a marker interface, meaning it doesn't declare any methods. It simply signals to the Java runtime that objects of this class can be serialized.
2. **ObjectOutputStream:** This class is used to write objects to an output stream in serialized form.
3. **ObjectInputStream:** This class is used to read objects from an input stream that contains serialized data.

Key Points:

1. **Serializable Interface**: A class must implement the java.io.Serializable interface to enable serialization.
2. **transient Keyword**: Fields marked as transient are not serialized. This is useful for fields that are not part of the object's persistent state (e.g., cached values, thread-specific data).
3. **serialVersionUID**:
    - This is a static final long field that acts as a version identifier for a serializable class
    - A unique identifier used during deserialization to verify that the sender and receiver of the serialized object are compatible.

    - If not declared, it is generated automatically by JVM, but it's a good practice to declare it explicitly
4. **Security:** Be cautious when deserializing data from untrusted sources, as it can pose security risks (deserialization vulnerabilities).

Use Cases:

- **Persistence:** Saving application state to a file so it can be restored later.
- **Network Communication:** Sending objects over a network between different systems (e.g., using Java RMI).
- **Caching:** Storing objects in memory for faster access.
- **Session Management:** Storing user session data in web applications.

7. New IO (NIO)

In Java, **Channels**, **Buffers**, and **Selectors** are key concepts introduced in the **New I/O (NIO)** package, which provides a more efficient and flexible way to perform input and output operations compared to the traditional I/O APIs.

1. Buffers

What they are:

Buffers are essentially blocks of memory used for holding data that is being read from or written to a channel. They replace the byte streams and character streams of traditional I/O.

- **Key properties:**
    - **Capacity:** The maximum amount of data the buffer can hold.
    - **Position:** The current position for reading or writing.
    - **Limit:** The index of the first element that should *not* be read or written.
- **Common Buffer Types:**
    - ByteBuffer: For byte data.
    - CharBuffer: For character data.

- ShortBuffer, IntBuffer, LongBuffer, FloatBuffer, DoubleBuffer: For primitive data types.

Buffer Methods

Method	Description
`clear()`	Clears the buffer for a new write operation.
`flip()`	Prepares the buffer for a read operation.
`rewind()`	Resets position to zero without clearing data.
`get()`	Reads data from the buffer.
`put()`	Writes data into the buffer.
`remaining()`	Returns the number of elements between position and limit.

2. Channels

A **Channel** represents a connection to an entity capable of performing I/O operations. such as:

- Files (FileChannel)
- Sockets (SocketChannel, ServerSocketChannel, DatagramChannel)

Unlike streams, channels are bidirectional and can be used for both reading and writing.

Key characteristics:

- Channels are the source or destination of data for buffers.
- Data is read from a channel into a buffer, and data is written from a buffer to a channel.

Key Channel Types:

- FileChannel: For reading and writing files.
- SocketChannel: For TCP client sockets.
- ServerSocketChannel: For listening for incoming TCP connections.
- DatagramChannel: For UDP communication.

3. Selectors

A **Selector** is a Java NIO component that provides a mechanism for monitoring multiple channels (e.g., SocketChannel) for readiness events, such as readiness for reading, writing, or accepting a connection. This is particularly useful for multiplexed, non-blocking I/O operations.

- **How they work:**
    1. You create a Selector.
    2. You register one or more channels with the selector, specifying the I/O operations you're interested in (e.g., OP_READ, OP_WRITE, OP_CONNECT, OP_ACCEPT).
    3. You call the select() method on the selector. This method blocks until one or more registered channels are ready for an I/O operation.
    4. The select() method returns a set of SelectionKey objects, each representing a channel that is ready.
    5. You iterate through the selected keys and perform the appropriate I/O operations on the corresponding channels.

Relationship Between Channels, Buffers, and Selectors:

- Data is always transferred between a channel and a buffer.
- A selector allows a single thread to monitor multiple channels for I/O readiness.

Key Methods

Method	Description
`select()`	Blocks until at least one channel is ready.
`selectNow()`	Non-blocking version of `select()`.
`selectedKeys()`	Returns a set of keys for the channels ready for I/O operations.

8. File and Directory Operations

Java provides robust support for performing various file and directory operations through the java.io.File class (older approach)

and the java.nio.file package (New I/O or NIO.2, the more modern and preferred approach).

1. Using java.io.File (Older Approach):

Common Operations on Files and Directories

Operation	Description	Key Classes/Methods
Create a file	Create a new file	`createNewFile()`
Create a directory	Create a new directory	`mkdir()`
Check existence	Check if a file/directory exists	`exists()`
Delete a file	Delete a file or directory	`delete()`
List files in a folder	Get a list of files and subdirectories in a directory	`list()`
Get file attributes	Fetch details like size, name, and permissions	`length()`, `getName()`

2. Using java.nio.file (NIO.2 - Modern Approach):

The java.nio.file package provides a more comprehensive and flexible API for file and directory operations. Key classes and interfaces include:

- **Path:** Represents a file or directory path.
- **Files:** Provides static methods for performing various file and directory operations.

Key Advantages of NIO.2:

- Improved performance (especially for large files).
- More powerful and flexible API.
- Better support for symbolic links and other file system features.
- More robust error handling.

Common Operations in NewIO

Operation	Description	Key Classes/Methods
Create a file	Create a new file	`Files.createFile()`
Create a directory	Create a new directory	`Files.createDirectory()`, `Files.createDirectories()`, `Files.createDirectories()`
Check existence	Check if a file/directory exists	`File.exists()`, `Files.isDirectory()`, `Files.isRegularFile()`
Delete a file	Delete a file or directory	`Files.delete()`, `Files.deleteIfExists()`
List files in a folder	Get a list of files and subdirectories in a directory	`Files.list()`
Get file attributes	Fetch details like size, name, and permissions	`Files.readAttributes()`

9. Java Networking

Java provides a comprehensive networking API to facilitate communication between systems. The key components include Sockets, URLs, and InetAddress. They work together to enable communication between applications over a network. These tools allow developers to create network applications for tasks like sending data, retrieving web pages, or resolving host information.

1. Sockets

Sockets enable communication between two machines over a network. Sockets are the endpoints of a network connection. They represent a communication channel between two applications. Think of them as the "plugs" that connect two devices on a network. Java provides two main types of sockets:

- **TCP (Transmission Control Protocol)**: Reliable, connection-oriented communication.
- **UDP (User Datagram Protocol)**: Unreliable, connectionless communication.

Socket Classes

- **Socket:** For client-side communication using TCP.

- **ServerSocket:** For server-side communication using TCP.
- **DatagramSocket:** For both client and server communication using UDP.

Key Operations:

- **Client Side (Socket):**
  - Socket(String host, int port): Creates a socket and connects to the specified host and port.
  - getInputStream(): Returns an InputStream for reading data from the socket.
  - getOutputStream(): Returns an OutputStream for writing data to the socket.
  - close(): Closes the socket.
- **Server Side (ServerSocket):**
  - ServerSocket(int port): Creates a server socket that listens on the specified port.
  - accept(): Accepts an incoming client connection and returns a new Socket object for communication with that client.
  - close(): Closes the server socket.

2. URL

The java.net.URL class is used to represent a Uniform Resource Locator. A URL is a string that specifies the location of a resource on the internet. It typically includes the protocol (e.g., http, https), the hostname (or IP address), and the path to the resource.

java.net.URL Class: Provides methods for parsing and accessing URLs.

Key Methods

Method	Description
`URL(String url)`	Creates a URL object from a string
`openConnection()`	Opens a connection to the URL.
`getHost()`	Returns the host name of the URL.

Method	Description
`getPath()`	Returns the file path of the URL.
`getProtocol()`	Returns the protocol (e.g., HTTP, HTTPS).
`openStream()`	Opens an input stream to read data.

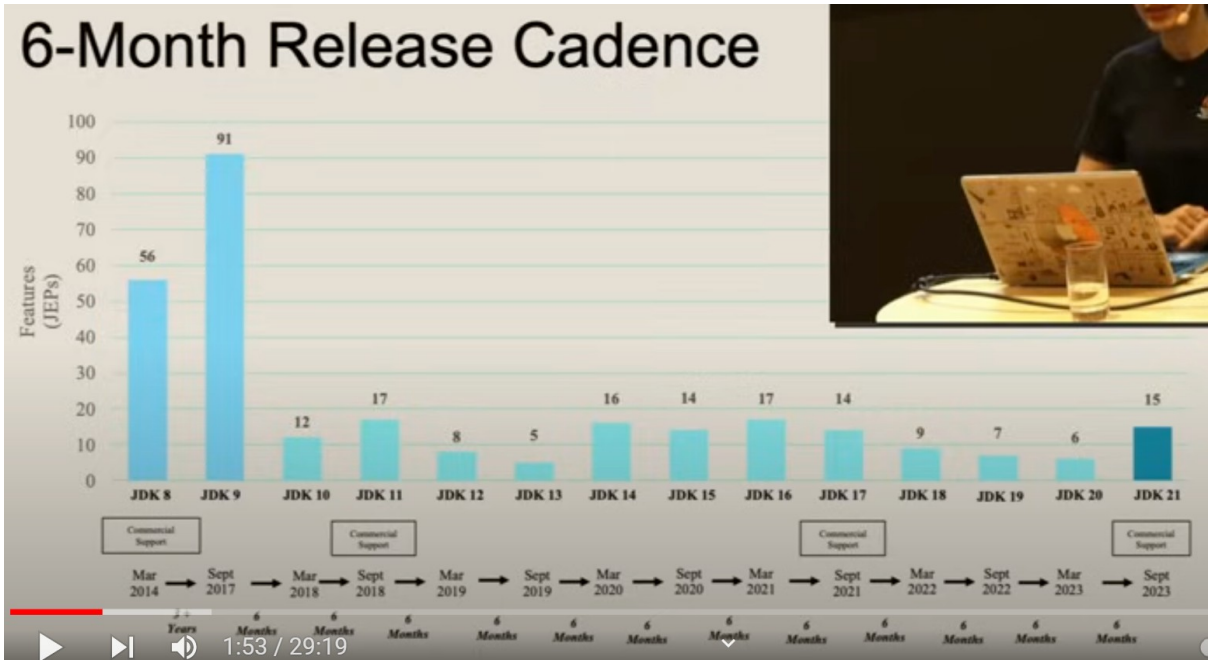
3. InetAddress

 What it is: Represents an IP address (either IPv4 or IPv6).
 `java.net.InetAddress` Class: Provides methods for getting IP addresses from hostnames and vice versa.

Key Methods

Method	Description
`getByName(String host)`	Returns the `InetAddress` of the given hostname.
`getHostAddress()`	Returns the IP address as a string.
`getHostName()`	Returns the hostname.
`getLocalHost()`	Returns the local host's IP address.

20. 6-Month Release Cadence



21. Java 8 , 11 and 17 Features

Please Join in below link

Instagram: https://www.instagram.com/rba_infotech/
LinkedIn: https://www.linkedin.com/company/rba-infotech/
Facebook:
https://www.facebook.com/people/RBA-Infotech/100043552406579/

Java 8	Java 11	Java 17
Lambda Expression	String methods IsEmpty(), isBlank(), strip(), stripLeading(), stripTrailing()	Indent(4), indent(-2) method
Functional Interface	toArray() to convert collection to Array	Transform() method
Default Method in Interface	private method in interface	Collectors.teeing() method
Static Method in Interface	writeStrin(), readString() method	toList() method
Method Reference	HttpClient	Text Block (“””) three double codes make JSON string stored in variable more readable
ForEach() method	List.copyOf() method	
Stream API filter(), map(), reduce(), min(), max(), count(), findFirst()	Predicate.not() method	
Date and Time API	orElseThrow() method	
Optional class	try-with-resource method	
Concurrency API		

22. Java Date Time API

1. Working with LocalDate, LocalTime, LocalDateTime

4. Working with LocalDate, LocalTime, and LocalDateTime in Java

The LocalDate, LocalTime, and LocalDateTime classes, introduced in Java 8 as part of the java.time package, are used for handling dates, times, and combined date-time values without timezone information. These classes are immutable and thread-safe.

---

5. 1. LocalDate

- Represents a date (year, month, day) without a time or timezone.
- Commonly used for calendar-related operations.

Creating LocalDate

```
import java.time.LocalDate;


public class LocalDateExample {
    public static void main(String[] args) {
        // Current date
        LocalDate today = LocalDate.now();
        System.out.println("Today: " + today);


        // Specific date
        LocalDate specificDate = LocalDate.of(2025, 1, 1);
        System.out.println("Specific Date: " + specificDate);


        // Parsing date from a string
        LocalDate parsedDate = LocalDate.parse("2025-01-01");
        System.out.println("Parsed Date: " + parsedDate);
    }
```

}

Common Methods

Method	Description
getYear()	Returns the year.
getMonth()	Returns the month.
getDayOfWeek()	Returns the day of the week.
plusDays(n)	Adds n days to the date.
minusMonths(n)	Subtracts n months from the date.
isLeapYear()	Checks if the year is a leap year.

6. 2. LocalTime

- Represents a time (hour, minute, second, nanosecond) without a date or timezone.
- Commonly used for operations related to time of day.

Creating LocalTime

```
import java.time.LocalTime;


public class LocalTimeExample {
    public static void main(String[] args) {
        // Current time
        LocalTime now = LocalTime.now();
        System.out.println("Current Time: " + now);


        // Specific time
        LocalTime specificTime = LocalTime.of(14, 30, 15);
        System.out.println("Specific Time: " + specificTime);
```

```
        // Parsing time from a string
        LocalTime parsedTime = LocalTime.parse("14:30:15");
        System.out.println("Parsed Time: " + parsedTime);
    }
}
```

Common Methods

Method	Description
getHour()	Returns the hour.
getMinute()	Returns the minute.
plusHours(n)	Adds n hours to the time.
minusMinutes (n)	Subtracts n minutes from the time.
isBefore()	Checks if a time is before another.

7. 3. LocalDateTime

- Combines LocalDate and LocalTime into a single object.
- Represents a date-time without timezone information.

Creating LocalDateTime

```
import java.time.LocalDateTime;

public class LocalDateTimeExample {
    public static void main(String[] args) {
        // Current date and time
        LocalDateTime now = LocalDateTime.now();
        System.out.println("Current DateTime: " + now);

        // Specific date and time
        LocalDateTime specificDateTime = LocalDateTime.of(2025, 1, 1, 14, 30, 15);
```

```
        System.out.println("Specific DateTime: " + specificDateTime);


        // Parsing date-time from a string
        LocalDateTime parsedDateTime = LocalDateTime.parse("2025-01-01T14:30:15");
        System.out.println("Parsed DateTime: " + parsedDateTime);
    }
}
```

Common Methods

Method	Description
toLocalDate()	Extracts the LocalDate component.
toLocalTime()	Extracts the LocalTime component.
plusWeeks(n)	Adds n weeks to the date-time.
minusSeconds(n)	Subtracts n seconds from the date-time.
format()	Formats the date-time using a pattern.

8. Formatting and Parsing

The DateTimeFormatter class is used to format and parse date-time values.

Formatting Example

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;


public class FormattingExample {
    public static void main(String[] args) {
        LocalDateTime now = LocalDateTime.now();
```

```
        DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");
        String formattedDateTime = now.format(formatter);
        System.out.println("Formatted DateTime: " +
formattedDateTime);
    }
}
```

Parsing Example

```
LocalDateTime parsedDateTime = LocalDateTime.parse("01-01-
2025 14:30:15", formatter);
```

---

9. Comparison and Operations

1. **Comparison**:
    - Use isBefore(), isAfter(), or isEqual() to compare dates or times.
2. **Duration and Period**:
    - Duration measures time-based amounts (e.g., hours, minutes).
    - Period measures date-based amounts (e.g., years, months, days).

Example:

```
import java.time.LocalDate;
import java.time.Period;

public class PeriodExample {
    public static void main(String[] args) {
        LocalDate startDate = LocalDate.of(2025, 1, 1);
        LocalDate endDate = LocalDate.of(2025, 12, 31);

        Period period = Period.between(startDate, endDate);
        System.out.println("Years: " + period.getYears());
```

```
        System.out.println("Months: " + period.getMonths());
        System.out.println("Days: " + period.getDays());
    }
}
```

---

10. Best Practices

1. Use LocalDate and LocalTime instead of java.util.Date for clearer and more maintainable code.
2. For timezone-related operations, use ZonedDateTime or OffsetDateTime.
3. Leverage DateTimeFormatter for custom formatting.
4. Avoid manual parsing and formatting; let the java.time package handle it for you.

Would you like to dive deeper into timezone handling or advanced operations?

11. Temporal Adjusters, Periods, and Time Zones

12. Temporal Adjusters, Periods, and Time Zones in Java

The java.time package, introduced in Java 8, provides powerful tools for manipulating dates, times, and time zones. This includes TemporalAdjuster, Period, and timezone-related classes like ZonedDateTime and OffsetDateTime.

---

23. 1. Temporal Adjusters

A TemporalAdjuster is a functional interface that allows you to modify Temporal objects like LocalDate or LocalDateTime in a user-defined way.

13. Common Use Cases

- Adjusting a date to the next Monday.
- Finding the last day of the current month.
- Setting the date to the first day of the next year.

14. Built-in Adjusters

The TemporalAdjusters utility class provides predefined adjusters.

Examples

```
import java.time.LocalDate;
import java.time.temporal.TemporalAdjusters;
import java.time.DayOfWeek;

public class TemporalAdjustersExample {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();

        // Adjust to the first day of the next month
        LocalDate firstDayNextMonth =
today.with(TemporalAdjusters.firstDayOfNextMonth());
        System.out.println("First Day of Next Month: " +
firstDayNextMonth);

        // Adjust to the next Monday
        LocalDate nextMonday =
today.with(TemporalAdjusters.next(DayOfWeek.MONDAY));
        System.out.println("Next Monday: " + nextMonday);

        // Adjust to the last day of the current month
        LocalDate lastDayOfMonth =
today.with(TemporalAdjusters.lastDayOfMonth());
        System.out.println("Last Day of the Month: " +
lastDayOfMonth);
    }
}
```

15. Custom Adjuster

You can create a custom TemporalAdjuster using a lambda or a class.

```
import java.time.LocalDate;
import java.time.temporal.TemporalAdjuster;
import java.time.temporal.Temporal;

public class CustomAdjusterExample {
    public static void main(String[] args) {
        TemporalAdjuster nextOddDay = temporal -> {
            LocalDate date = LocalDate.from(temporal);
            return (date.getDayOfMonth() % 2 == 0) ? date.plusDays(1) : date;
        };

        LocalDate today = LocalDate.now();
        LocalDate nextOddDate = today.with(nextOddDay);
        System.out.println("Next Odd Day: " + nextOddDate);
    }
}
```

---

24. 2. Periods

A Period represents a quantity of time in terms of years, months, and days. It is useful for date-based calculations.

16. Creating a Period

```
import java.time.LocalDate;
import java.time.Period;

public class PeriodExample {
    public static void main(String[] args) {
```

```
        LocalDate startDate = LocalDate.of(2023, 1, 1);
        LocalDate endDate = LocalDate.of(2025, 1, 1);

        // Calculate the period between two dates
        Period period = Period.between(startDate, endDate);

        System.out.println("Years: " + period.getYears());
        System.out.println("Months: " + period.getMonths());
        System.out.println("Days: " + period.getDays());
    }
}
```

17. Manipulating Dates with Period

```
LocalDate today = LocalDate.now();
Period period = Period.of(1, 2, 15); // 1 year, 2 months, 15 days
LocalDate adjustedDate = today.plus(period);
System.out.println("Adjusted Date: " + adjustedDate);
```

---

25. 3. Time Zones

Java provides several classes for handling time zones:

- ZonedDateTime: Combines date, time, and timezone.
- OffsetDateTime: Includes a time offset (e.g., +05:30).
- ZoneId: Represents a specific time zone (e.g., Asia/Kolkata).

18. Working with Time Zones

Getting the Current Time in a Specific Time Zone

```
import java.time.ZonedDateTime;
import java.time.ZoneId;

public class ZonedDateTimeExample {
    public static void main(String[] args) {
```

```
        ZonedDateTime zonedDateTime =
ZonedDateTime.now(ZoneId.of("Asia/Kolkata"));
        System.out.println("Current Time in Asia/Kolkata: " +
zonedDateTime);
    }
}
```

Converting Between Time Zones

```
import java.time.ZonedDateTime;
import java.time.ZoneId;

public class TimeZoneConversion {
    public static void main(String[] args) {
        ZonedDateTime utcTime =
ZonedDateTime.now(ZoneId.of("UTC"));
        ZonedDateTime localTime =
utcTime.withZoneSameInstant(ZoneId.of("Asia/Kolkata"));

        System.out.println("UTC Time: " + utcTime);
        System.out.println("Local Time (Asia/Kolkata): " + localTime);
    }
}
```

---

19. **4. OffsetDateTime**

An OffsetDateTime represents a date-time with an offset from UTC.

Example

```
import java.time.OffsetDateTime;
import java.time.ZoneOffset;

public class OffsetDateTimeExample {
    public static void main(String[] args) {
```

```
        OffsetDateTime offsetDateTime =
OffsetDateTime.now(ZoneOffset.of("+05:30"));
        System.out.println("Offset Date-Time: " + offsetDateTime);
    }
}
```

---

20. 5. Best Practices

1. **Use ZoneId for Accurate Time Zones**:
    - Prefer using ZoneId (e.g., Asia/Kolkata) over fixed offsets for better daylight saving adjustments.
2. **Handle Time Zones Explicitly**:
    - Always specify a timezone when dealing with global applications to avoid confusion.
3. **Leverage Adjusters for Date Manipulations**:
    - Use built-in or custom TemporalAdjuster for readable and maintainable date adjustments.
4. **Combine Periods and Durations**:
    - Use Period for date differences and Duration for time differences.

Would you like to explore more advanced topics like Duration, Instant, or ChronoUnit?

26. Functional Interfaces

1. Functional Programming Concepts

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It emphasizes immutability, pure functions, and the application of functions to arguments. Here are some core concepts:

1. Pure Functions:

- A pure function always produces the same output for the same input and has no side effects.
- **No Side Effects:** It doesn't modify any external state (variables, objects, I/O).
- **Referential Transparency:** Replacing a function call with its result (or vice versa) doesn't change the program's behavior. This makes reasoning about code and testing much easier.

Java

```
// Pure function
int add(int a, int b) {
    return a + b;
}

// Impure function (has side effect)
int counter = 0;
int increment() {
    return ++counter; // Modifies external state
}
```

2. Immutability:

- Data is immutable, meaning its value cannot be changed after it is created.
- To "modify" data, you create a new copy with the desired changes.
- This prevents many common concurrency issues and makes code easier to reason about.

Java

```
// Example (using Java's String, which is immutable)
String str = "hello";
String newStr = str.toUpperCase(); // Creates a new string "HELLO"
System.out.println(str); // Output: hello (original string unchanged)
System.out.println(newStr); // Output: HELLO
```

3. First-Class Functions:

- Functions can be treated as values: passed as arguments to other functions, returned as results from functions, and assigned to variables.

Java

```
import java.util.function.Function;


Function<Integer, Integer> square = x -> x * x; // Assigning a lambda to a variable

int result = square.apply(5); // Calling the function
```

4. Higher-Order Functions:

- Functions that take other functions as arguments or return functions as results.
- Common examples include map, filter, and reduce.

```
import java.util.Arrays; import java.util.List; import java.util.stream.Collectors;

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);


// Using map (a higher-order function) to square each number

List<Integer> squaredNumbers = numbers.stream()

        .map(x -> x * x)

        .collect(Collectors.toList());
```

```

**5. Lambda Expressions (Anonymous Functions):**

- Short, concise way to define anonymous functions (functions without a name).

Java

```
// Lambda expression in Java

(int x, int y) -> x + y;
```

**6. Recursion:**

- Using functions that call themselves to solve problems. Often used instead of loops in functional programming.

Java
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

7. Lazy Evaluation:

- Expressions are not evaluated until their results are needed. This can improve performance by avoiding unnecessary computations. Java Streams support lazy evaluation.

Java

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);


// Only the first 3 squared numbers will be calculated
numbers.stream()
        .map(x -> {
            System.out.println("Squaring " + x); // This will only print for 1, 2, and 3
            return x * x;
        })
        .limit(3)
        .collect(Collectors.toList());
```

Benefits of Functional Programming:

- **Improved Code Readability and Maintainability:** Pure functions and immutability make code easier to understand and reason about.
- **Easier Testing:** Pure functions are easier to test because their output depends only on their input.
- **Concurrency and Parallelism:** Immutability eliminates many common concurrency problems, making it easier to write parallel code.

- **Reduced Bugs:** Avoiding side effects and mutable state can lead to fewer bugs.

While Java is primarily an object-oriented language, it has incorporated many functional programming features (especially with the introduction of lambda expressions and streams in Java 8), making it possible to write code with a more functional style.

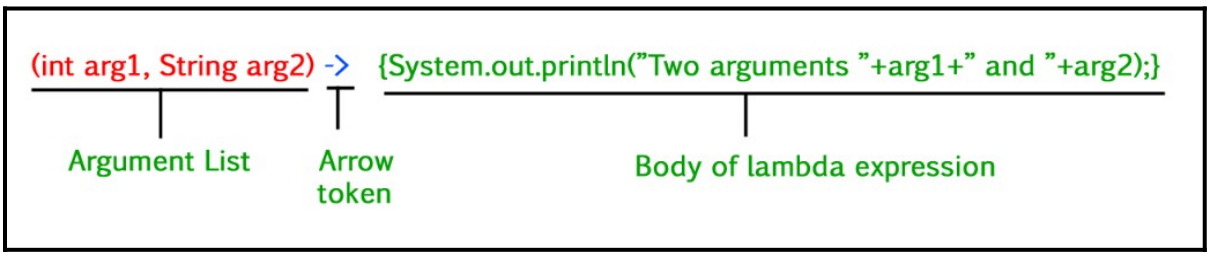
27. Lambda Expression

21. Lambda Expressions: Syntax and Usage

ChatGPT Result

22. Lambda Expressions in Java: Syntax and Usage

In Java, lambda expressions were introduced in **Java 8** to enable functional programming and make the code more concise. A lambda expression in Java allows you to write inline implementations of functional interfaces (interfaces with a single abstract method).



---

23. Syntax

(parameters) -> expression

// Or for multiple statements:

(parameters) -> {

    // statements

    return value;

}

- **Parameters**: The input arguments for the lambda expression.

- **Arrow (->)**: Separates the parameters from the body of the lambda.
- **Expression/Body**: The logic to execute; can be a single expression or a block of statements.

---

24. Key Features

1. **Concise**: Removes boilerplate code for anonymous class implementations.
2. **Functional Interface**: Can only be used with functional interfaces.
3. **Improved Readability**: Simplifies the syntax for short operations.

---

25. Examples of Lambda Expressions

1. Basic Lambda Expression

For a functional interface like this:

```
@FunctionalInterface
interface MathOperation {
    int operate(int a, int b);
}
```

You can implement it with a lambda:

```
MathOperation addition = (a, b) -> a + b;
System.out.println(addition.operate(5, 3)); // Output: 8
```

---

2. Using Lambda with a Single Parameter

For an interface like this:

```
@FunctionalInterface
interface Greeting {
    void sayHello(String name);
}
```

You can implement it:

```
Greeting greet = name -> System.out.println("Hello, " + name);
greet.sayHello("Alice"); // Output: Hello, Alice
```

---

3. Multiple Statements in Lambda

If the body has multiple statements, use curly braces {}.

```
MathOperation multiplication = (a, b) -> {
    int result = a * b;
    System.out.println("Multiplying " + a + " and " + b);
    return result;
};
System.out.println(multiplication.operate(4, 5)); // Output: Multiplying 4 and 5, 20
```

---

26. Common Usage in Java

1. Using Lambda with Built-in Functional Interfaces

Java provides several functional interfaces in the java.util.function package, such as Predicate, Consumer, Function, and Supplier.

- **Predicate<T>**: Evaluates a condition and returns true or false.
- Predicate<Integer> isEven = x -> x % 2 == 0;
- System.out.println(isEven.test(4)); // Output: true
- **Consumer<T>**: Performs an operation on a single input.
- Consumer<String> print = x -> System.out.println(x);
- print.accept("Hello World!"); // Output: Hello World!
- **Function<T, R>**: Takes an input and produces an output.
- Function<Integer, String> toString = x -> "Number: " + x;
- System.out.println(toString.apply(5)); // Output: Number: 5
- **Supplier<T>**: Supplies a value.
- Supplier<Double> random = () -> Math.random();

- System.out.println(random.get()); // Output: (some random value)

---

2. Using Lambda with Collections

- **forEach in Lists**
- List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
- names.forEach(name -> System.out.println(name));
- **Comparator for Sorting**
- List<Integer> numbers = Arrays.asList(5, 2, 8, 1);
- Collections.sort(numbers, (a, b) -> a - b); // Ascending order
- System.out.println(numbers); // Output: [1, 2, 5, 8]

---

3. Using Lambda with Streams

Streams API in Java 8 is highly compatible with lambda expressions.

- **Filter and Map**
- List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
- List<Integer> squared = numbers.stream()
- .map(x -> x * x)
- .collect(Collectors.toList());
- System.out.println(squared); // Output: [1, 4, 9, 16, 25]
- **Reduce**
- int sum = numbers.stream()
- .reduce(0, (a, b) -> a + b);
- System.out.println(sum); // Output: 15

---

27. Advantages of Lambda Expressions in Java

1. **Reduces Boilerplate Code**: Simplifies functional interface implementations.
2. **Improves Readability**: Code becomes concise and easier to understand.

3. **Functional Programming**: Enables features like immutability and declarative coding.

---

28. Limitations of Lambda Expressions

1. **Readability Issues**: Overuse or complex lambdas can reduce clarity.
2. **Debugging Challenges**: Debugging lambda expressions is harder than regular methods.
3. **Single Abstract Method**: Limited to functional interfaces.

Would you like more examples, or should I explain specific functional interfaces or stream operations?

Gemini Result

Lambda expressions, introduced in Java 8, provide a concise way to represent anonymous functions (functions without a name). They enable you to treat functionality as a method argument, or code as data.

Syntax:

A lambda expression consists of three parts:

1. **Parameter List:** The input parameters to the function. Can be empty, a single parameter, or multiple parameters.
2. **Arrow Token (->):** Separates the parameter list from the lambda body.
3. **Lambda Body:** The code that implements the function. Can be a single expression or a block of code enclosed in curly braces {}.

Here's the general syntax:

Java

(parameter1, parameter2, ...) -> { body }

Examples:

- **No parameters:**

Java

() -> System.out.println("Hello, Lambda!");

- **Single parameter:**

Java

```
x -> x * x; // Returns the square of x
(int x) -> x * x; // Explicitly declaring the parameter type
```

- **Multiple parameters:**

Java

```
(int x, int y) -> x + y; // Returns the sum of x and y
(String s1, String s2) -> s1.concat(s2); // Concatenates two strings
```

- **Lambda body with a block of code:**

Java

```
(int x) -> {
    int result = x * 2;
    System.out.println("Result: " + result);
    return result; // Explicit return statement required in block body
};
```

Usage:

Lambda expressions are primarily used in the context of *functional interfaces*.

Functional Interfaces:

A functional interface is an interface that contains exactly one abstract method. They can have default methods and static methods, but only one abstract method. The @FunctionalInterface annotation can be used to explicitly mark an interface as functional, and the compiler will enforce this rule.

Examples of built-in functional interfaces in Java:

- Runnable: void run()
- Callable<V>: V call()
- Consumer<T>: void accept(T t)
- Function<T, R>: R apply(T t)
- Predicate<T>: boolean test(T t)
- Supplier<T>: T get()

Using Lambdas with Functional Interfaces:

You can assign a lambda expression to a variable of a functional interface type. The lambda's signature (parameter types and return type) must match the signature of the functional interface's abstract method.

Java

```
import java.util.function.Function;


public class LambdaExample {
    public static void main(String[] args) {
        // Using a lambda with the Function interface
        Function<Integer, Integer> square = x -> x * x;
        int result = square.apply(5); // result will be 25
        System.out.println(result);


        // Using a lambda with the Runnable interface
        Runnable myRunnable = () -> System.out.println("Running in a thread!");
        new Thread(myRunnable).start();


        // Using a lambda with the Predicate Interface
        java.util.function.Predicate<Integer> isEven = n -> n % 2 == 0;
        System.out.println(isEven.test(4)); // true
        System.out.println(isEven.test(7)); // false



    }
}
```

Method References:

Method references are a shorthand notation for lambda expressions that simply call an existing method. There are four types of method references:

1. **Reference to a static method:** ClassName::staticMethodName
2. **Reference to an instance method of a particular object:** objectReference::instanceMethodName
3. **Reference to an instance method of an arbitrary object of a particular type:** ClassName::instanceMethodName [1]

1. github.com

github.com

4. **Reference to a constructor:** ClassName::new

Java

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;

public class MethodReferenceExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        // Using a lambda expression
        names.forEach(name -> System.out.println(name));

        // Using a method reference (equivalent to the lambda above)
        names.forEach(System.out::println);
```

```
        // Example of a static method reference
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        numbers.forEach(MethodReferenceExample::printNumber);
    }

    public static void printNumber(int number) {
        System.out.println("The Number is " + number);
    }

}
```

Lambda expressions and method references significantly enhance Java's functional programming capabilities, making code more concise and expressive, especially when working with collections and streams.

29. Functional Interfaces: Predicate, Function, Consumer, Supplier

ChatGPT Result

30. Functional Interfaces in Java

A **functional interface** is an interface with exactly **one abstract method**, allowing its implementation using **lambda expressions** or **method references**. Functional interfaces are key components in Java's functional programming features introduced in Java 8.

The most commonly used functional interfaces are provided in the java.util.function package, such as:

1. **Predicate**
2. **Function**
3. **Consumer**
4. **Supplier**

---

31. **1. Predicate**

A Predicate is a functional interface that represents a **boolean-valued function** of one argument. It is typically used for conditional checks.

Abstract Method:

boolean test(T t);

Usage:

- Testing conditions
- Filtering data in streams

Example:

```
import java.util.function.Predicate;

public class PredicateExample {
    public static void main(String[] args) {
        Predicate<Integer> isEven = num -> num % 2 == 0;

        System.out.println(isEven.test(4));  // Output: true
        System.out.println(isEven.test(7));  // Output: false
    }
}
```

Key Methods in Predicate:

- and(): Combines two predicates with a logical AND.
- or(): Combines two predicates with a logical OR.
- negate(): Returns the negation of the predicate.

Example:

```
Predicate<Integer> isPositive = num -> num > 0;
Predicate<Integer> isNegative = isPositive.negate();

System.out.println(isNegative.test(-5));  // Output: true
```

---

32. 2. Function

A Function is a functional interface that represents a function that takes one argument and produces a result.

Abstract Method:

R apply(T t);

Usage:

- Mapping data
- Transforming data

Example:

```
import java.util.function.Function;


public class FunctionExample {
    public static void main(String[] args) {
        Function<String, Integer> stringLength = str -> str.length();


        System.out.println(stringLength.apply("Hello"));  // Output: 5
    }
}
```

Key Methods in Function:

- andThen(): Chains two functions where the result of one function is passed to another.
- compose(): Chains two functions where the result of the second function is passed to the first.

Example:

```
Function<Integer, Integer> square = x -> x * x;
Function<Integer, Integer> doubleValue = x -> x * 2;


System.out.println(square.andThen(doubleValue).apply(3));  // Output: 18
System.out.println(square.compose(doubleValue).apply(3));  // Output: 36
```

---

33. 3. Consumer

A Consumer is a functional interface that represents an operation that accepts a single input argument and returns no result (void).

Abstract Method:

void accept(T t);

Usage:

- Performing actions on data
- Iterating over data

Example:

```
import java.util.function.Consumer;

public class ConsumerExample {
    public static void main(String[] args) {
        Consumer<String> print = str -> System.out.println("Hello, " + str);

        print.accept("Alice");  // Output: Hello, Alice
    }
}
```

Key Method in Consumer:

- andThen(): Chains two consumers to execute sequentially.

Example:

```
Consumer<String> greet = str -> System.out.print("Hello, ");
Consumer<String> name = str -> System.out.println(str);

greet.andThen(name).accept("Alice");  // Output: Hello, Alice
```

---

34. 4. Supplier

A Supplier is a functional interface that represents a supplier of results. It does not take any arguments and returns a result.

Abstract Method:

T get();

Usage:

- Generating data
- Providing default values

Example:

```
import java.util.function.Supplier;


public class SupplierExample {
    public static void main(String[] args) {
        Supplier<String> messageSupplier = () -> "Hello, World!";


        System.out.println(messageSupplier.get());  // Output: Hello, World!
    }
}
```

---

35. Comparison of Predicate, Function, Consumer, and Supplier

Functional Interface	Abstract Method Signature	Purpose	Example Usage
Predicate	boolean test(T t)	Represents a condition (returns a boolean).	Filtering a list of numbers.
Function	R apply(T t)	Transforms input into output.	Mapping data in streams.

Functional Interface	Abstract Method Signature	Purpose	Example Usage
Consumer	void accept(T t)	Performs an operation on input (no return).	Printing or saving data.
Supplier	T get()	Supplies a value (no input).	Generating random numbers.

36. Real-World Example

Using all functional interfaces together:

```
import java.util.function.*;

public class FunctionalInterfaceExample {
    public static void main(String[] args) {
        Predicate<Integer> isPositive = num -> num > 0;
        Function<Integer, String> intToString = num -> "Number: " + num;
        Consumer<String> print = str -> System.out.println(str);
        Supplier<Double> randomSupplier = () -> Math.random();

        // Test Predicate
        System.out.println(isPositive.test(10));  // Output: true

        // Use Function
        String result = intToString.apply(42);
        print.accept(result);                     // Output: Number: 42

        // Use Supplier
        System.out.println("Random Number: " + randomSupplier.get());
```

```
    }
}
```

This covers the basics of these functional interfaces. Let me know if you need more examples or advanced details! 😊

Gemini Result

Functional interfaces are a key component of functional programming in Java. They are interfaces with exactly one abstract method. This single abstract method defines the function's signature. Lambda expressions and method references can be used to provide implementations for these abstract methods.

Here's a breakdown of the commonly used functional interfaces: Predicate, Function, Consumer, and Supplier:

1. Predicate<T>:

- **Abstract Method:** boolean test(T t)
- **Purpose:** Represents a function that takes an argument of type T and returns a boolean value. Used for filtering or testing conditions.

Java

```
import java.util.function.Predicate;

public class PredicateExample {
    public static void main(String[] args) {
        Predicate<Integer> isEven = n -> n % 2 == 0;
        System.out.println(isEven.test(4)); // Output: true
        System.out.println(isEven.test(7)); // Output: false

        Predicate<String> isLong = s -> s.length() > 5;
        System.out.println(isLong.test("Hello")); // Output: false
        System.out.println(isLong.test("Welcome")); // Output: true

        // Combining Predicates
```

```
        Predicate<String> startsWithA = s -> s.startsWith("A");
        Predicate<String> longerThan4 = s -> s.length() > 4;

        Predicate<String> combinedPredicate =
startsWithA.and(longerThan4);
        System.out.println(combinedPredicate.test("Apple")); //true
        System.out.println(combinedPredicate.test("Ant")); //false
    }
}
```

2. Function<T, R>:

- **Abstract Method:** R apply(T t)
- **Purpose:** Represents a function that takes an argument of type T and returns a result of type R. Used for transforming or mapping values.

Java

```
import java.util.function.Function;

public class FunctionExample {
    public static void main(String[] args) {
        Function<String, Integer> stringLength = s -> s.length();
        System.out.println(stringLength.apply("Hello")); // Output: 5

        Function<Integer, String> intToString = i -> String.valueOf(i);
        System.out.println(intToString.apply(123)); // Output: "123"

        // Chaining Functions
        Function<String, String> toUpperCase = String::toUpperCase;
        Function<String, String> addExclamation = s -> s + "!";

        Function<String, String> combinedFunction =
toUpperCase.andThen(addExclamation);
```

```
        System.out.println(combinedFunction.apply("hello")); //HELLO!
    }
}
```

3. Consumer<T>:

- **Abstract Method:** void accept(T t)
- **Purpose:** Represents an operation that accepts an argument of type T and performs some action on it, but does not return a result (void). Used for performing side effects, such as printing to the console or modifying an object.

Java

```
import java.util.function.Consumer;
import java.util.List;
import java.util.Arrays;


public class ConsumerExample {
    public static void main(String[] args) {
        Consumer<String> printUpperCase = s -> System.out.println(s.toUpperCase());
        printUpperCase.accept("hello"); // Output: HELLO


        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        numbers.forEach(System.out::println); // Using method reference


        Consumer<Integer> printEvenOrOdd = n -> {
            if(n%2 == 0) System.out.println(n + " is Even");
            else System.out.println(n + " is Odd");
        };
        numbers.forEach(printEvenOrOdd);
    }
}
```

4. Supplier<T>:

- **Abstract Method:** T get()
- **Purpose:** Represents a function that does not take any arguments and returns a value of type T. Used for generating or providing values.

Java

```
import java.util.function.Supplier;
import java.util.Random;


public class SupplierExample {
    public static void main(String[] args) {
        Supplier<String> helloSupplier = () -> "Hello from Supplier!";
        System.out.println(helloSupplier.get()); // Output: Hello from Supplier!


        Supplier<Integer> randomIntSupplier = () -> new Random().nextInt(100); // 0 to 99
        System.out.println(randomIntSupplier.get());


    }
}
```

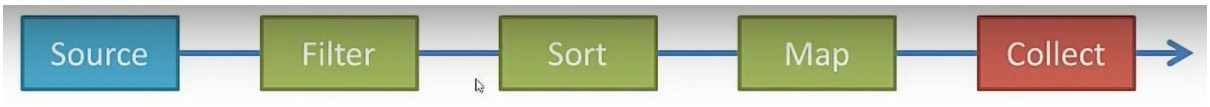
Key Points:

- These functional interfaces are part of the java.util.function package.
- They are heavily used with Java Streams API for performing operations on collections.
- Understanding these interfaces is crucial for effectively using lambda expressions and functional programming concepts in Java.

By using these standard functional interfaces, you can write more concise and expressive code, especially when dealing with collections, streams, and asynchronous operations. They also promote code reusability and improve readability.

28. Stream API

A Stream pipeline consist of source followed by zero or more intermediate operations, and a terminal operation



Stream Source:

Stream can be created from Collections, List, Set, Arrays, lines of file.

Stream Operations are either Intermediate or Terminal Operations

Intermediate Operations:

Such as following methods return a stream.

- filter()
- map()
- sorted()
- findFirst()
- anyMatch()
- distinct()

Terminal Operations:

Such as following methods returns a non-stream

- forEach()
- reduce()
- collect()
- count()
- min()
- max()

29. Default Methods

30. Static methods in Interface

31. Method References

1. Method References in Java

A **method reference** in Java is a shorthand notation for invoking methods using a functional interface. It is an elegant way to pass the reference of a method instead of writing a lambda expression.

Method references are introduced in Java 8 as part of the functional programming features.

---

2. Types of Method References

1. **Reference to a Static Method**
   Syntax: ClassName::staticMethodName
2. **Reference to an Instance Method of a Specific Object**
   Syntax: instance::instanceMethodName
3. **Reference to an Instance Method of an Arbitrary Object of a Particular Type**
   Syntax: ClassName::instanceMethodName
4. **Reference to a Constructor**
   Syntax: ClassName::new

---

1. Reference to a Static Method

You can refer to a static method directly using the class name.

Example:

```
import java.util.function.Consumer;

public class StaticMethodReference {
    public static void printMessage(String message) {
        System.out.println(message);
    }

    public static void main(String[] args) {
        Consumer<String> consumer =
StaticMethodReference::printMessage;
        consumer.accept("Hello, World!"); // Output: Hello, World!
    }
}
```

Lambda Equivalent:

```
Consumer<String> consumer = message ->
StaticMethodReference.printMessage(message);
```

---

2. Reference to an Instance Method of a Specific Object

You can refer to an instance method of a particular object.

Example:

```
import java.util.function.Consumer;

public class InstanceMethodReference {
    public void display(String message) {
        System.out.println(message);
    }

    public static void main(String[] args) {
        InstanceMethodReference obj = new
InstanceMethodReference();
```

```
        Consumer<String> consumer = obj::display;
        consumer.accept("Hello, Instance Method!"); // Output: Hello, Instance Method!
    }
}
```

Lambda Equivalent:

```
Consumer<String> consumer = message -> obj.display(message);
```

---

3. Reference to an Instance Method of an Arbitrary Object of a Particular Type

This type is useful when you want to call an instance method on objects being processed in a stream.

Example:

```
import java.util.function.Function;

public class ArbitraryInstanceMethodReference {
    public static void main(String[] args) {
        Function<String, Integer> lengthFunction = String::length;
        System.out.println(lengthFunction.apply("Hello")); // Output: 5
    }
}
```

Lambda Equivalent:

```
Function<String, Integer> lengthFunction = str -> str.length();
```

---

4. Reference to a Constructor

You can refer to a constructor to create objects dynamically.

Example:

```
import java.util.function.Supplier;

public class ConstructorReference {
```

```
    public ConstructorReference() {
        System.out.println("Constructor Invoked!");
    }

    public static void main(String[] args) {
        Supplier<ConstructorReference> supplier =
ConstructorReference::new;
        ConstructorReference obj = supplier.get(); // Output:
Constructor Invoked!
    }
}
```

Lambda Equivalent:

```
Supplier<ConstructorReference> supplier = () -> new
ConstructorReference();
```

---

5. Usage in Streams

Method references are often used in combination with the Java Streams API.

Example: Static Method Reference with Stream

```
import java.util.Arrays;
import java.util.List;

public class MethodReferenceWithStreams {
    public static void print(String str) {
        System.out.println(str);
    }

    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
        names.forEach(MethodReferenceWithStreams::print);
    }
```

}

Example: Instance Method Reference with Stream

```
import java.util.Arrays;
import java.util.List;

public class InstanceMethodReferenceStream {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
        names.stream()
            .map(String::toUpperCase) // Arbitrary instance method
            .forEach(System.out::println); // Static method reference
    }
}
```

---

6. Comparison of Method References and Lambda Expressions

Feature	Lambda Expression	Method Reference
Syntax	args -> method(args)	ClassName::methodName
Readability	May be verbose for simple operations	More concise and easier to read
Use Cases	Complex operations or inline logic	Directly invoking existing methods

7. Key Points to Remember

1. **Functional Interface Requirement**: Method references can only be used with functional interfaces.
2. **Replacement of Lambdas**: Use method references as a cleaner alternative when the lambda directly calls a method.
3. **Flexibility**: Can reference static, instance, or constructor methods.

Let me know if you'd like more examples or deeper explanations! 😊

32. Collection API Improvements

33. ForEach() method

34. Java IO Improvements

35. Optional Class

36. Concurrency API Improvements

37. JDBC Enhancements

38. Nashorn Javascript Engine

39. Base64Encode Decode

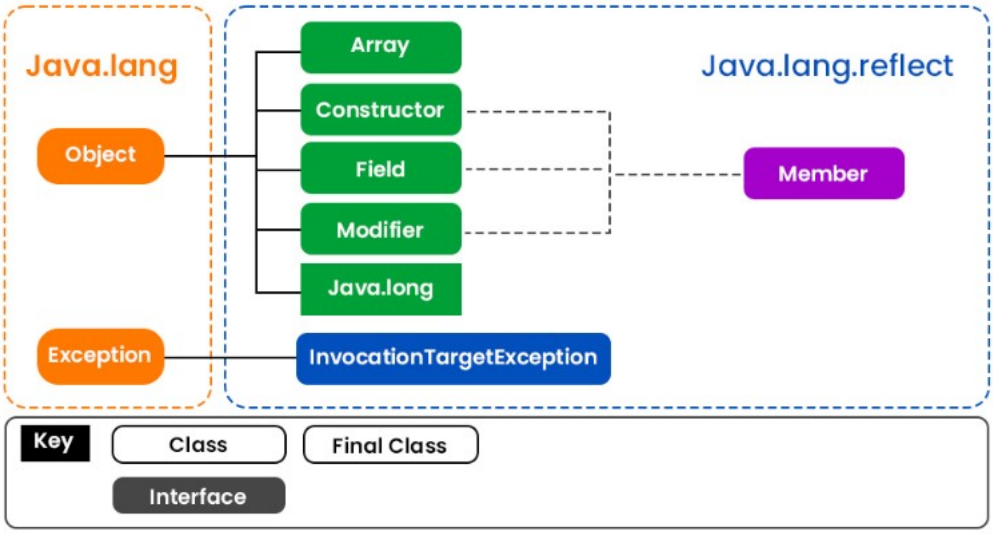
40. Parallel Array Sorting

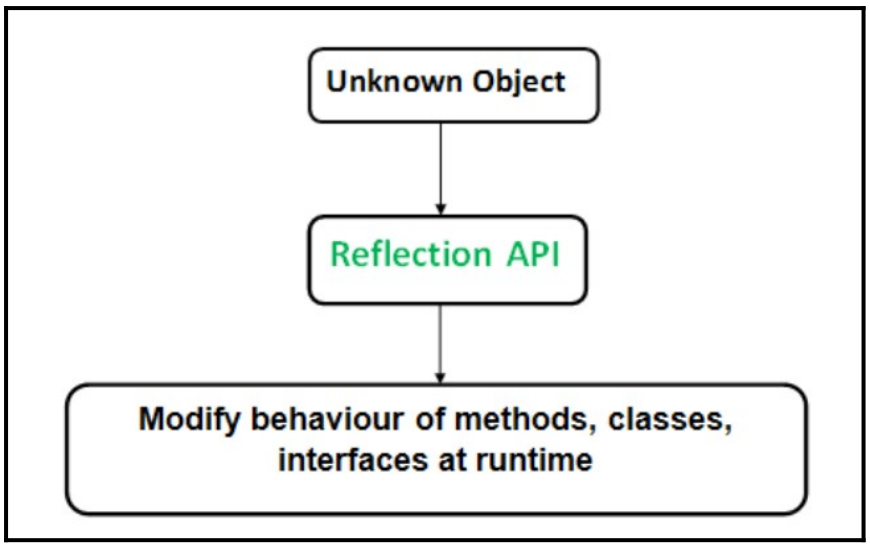
41. Java Reflection

1. Reflection API: Accessing and Modifying Runtime Classes

The Reflection API in Java is a powerful feature that allows you to inspect and manipulate classes, fields, methods, and constructors at runtime. This is particularly useful for dynamic behavior, debugging, testing frameworks, and tools like serialization libraries.

- Reflection gives us information about the class to which an object belongs and also the methods of that class that can be executed by using the object.
- Through reflection, we can invoke methods at runtime irrespective of the access specifier used with them.





Reflection can be used to get information about class, constructors, and methods as depicted below in tabular format as shown:

Class	The getClass() method is used to get the name of the class to which an object belongs.
Constructors	The getConstructors() method is used to get the public constructors of the class to which an object belongs.
Methods	The getMethods() method is used to get the public methods of the class to which an object belongs.

We can invoke a method through reflection if we know its name and parameter types. We use two methods for this purpose as described below before moving ahead as follows:

1. getDeclaredMethod()
2. invoke()

Method 1: getDeclaredMethod(): It creates an object of the method to be invoked.

Syntax: The syntax for this method

Class.getDeclaredMethod(name, parametertype)

Parameters:

- Name of a method whose object is to be created
- An array of Class objects

Method 2: invoke(): It invokes a method of the class at runtime we use the following method.

Syntax:

Method.invoke(Object, parameter)

Tip: *If the method of the class doesn't accept any parameter then null is passed as an argument.*

Note: *Through reflection, we can access the private variables and methods of a class with the help of its class object and invoke the method by using the object as discussed above. We use below two methods for this purpose.*

Method 3: Class.getDeclaredField(FieldName): Used to get the private field. Returns an object of type Field for the specified field name.

Method 4: Field.setAccessible(true): Allows to access the field irrespective of the access modifier used with the field.

1. Important observations Drawn From Reflection API

- **Extensibility Features:** An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.
- **Debugging and testing tools**: Debuggers use the property of reflection to examine private members of classes.
- **Performance Overhead:** Reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code that are called frequently in performance-sensitive applications.
- **Exposure of Internals:** Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

Programming Code: https://www.geeksforgeeks.org/reflection-in-java/

42. Annotations

1. Annotations: Built-in and Custom Annotations

https://www.javatpoint.com/java-annotation

43. JDBC Programming

1. Using PreparedStatement for efficient queries

2. Using PreparedStatement for Efficient Queries in JDBC

The PreparedStatement interface in Java provides a powerful way to execute parameterized SQL queries, improving performance and security compared to Statement. It is part of the java.sql package.

---

2. Advantages of PreparedStatement

1. **Performance**:
    - SQL queries are precompiled and stored, leading to faster execution for repeated queries.
    - Reduces database overhead since the query plan is reused.
2. **Security**:
    - Protects against SQL injection attacks by treating input parameters as data, not executable code.
3. **Ease of Use**:
    - Simplifies the handling of parameters, especially for complex queries or when working with data types like dates or blobs.
4. **Readability**:
    - Query templates with placeholders make the code easier to read and maintain.

---

3. Steps to Use PreparedStatement

1. **Create a Database Connection**:
   - Establish a connection to the database using DriverManager or a DataSource.
2. **Prepare the SQL Query**:
   - Use Connection.prepareStatement() to create a PreparedStatement object.
   - Use placeholders (?) for input parameters.
3. **Set Parameters**:
   - Bind values to placeholders using methods like setInt(), setString(), setDate(), etc.
4. **Execute the Query**:
   - Use methods like executeQuery() for SELECT queries or executeUpdate() for INSERT, UPDATE, and DELETE operations.
5. **Process Results**:
   - Use a ResultSet to retrieve and process query results for SELECT queries.
6. **Close Resources**:
   - Close ResultSet, PreparedStatement, and Connection objects to prevent resource leaks.

---

4. Example: Using PreparedStatement

1. Select Query:

```
import java.sql.*;


public class PreparedStatementExample {
    public static void main(String[] args) {
        String dbURL = "jdbc:mysql://localhost:3306/mydb";
        String username = "root";
        String password = "password";
```

```
        String query = "SELECT * FROM employees WHERE department
= ? AND salary > ?";


        try (Connection connection =
DriverManager.getConnection(dbURL, username, password);
             PreparedStatement preparedStatement =
connection.prepareStatement(query)) {


            // Set parameters
            preparedStatement.setString(1, "HR");
            preparedStatement.setDouble(2, 50000);


            // Execute query
            ResultSet resultSet = preparedStatement.executeQuery();


            // Process results
            while (resultSet.next()) {
                int id = resultSet.getInt("id");
                String name = resultSet.getString("name");
                double salary = resultSet.getDouble("salary");
                System.out.printf("ID: %d, Name: %s, Salary: %.2f%n", id,
name, salary);
            }


        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

---

2. Insert Query:

```
String insertQuery = "INSERT INTO employees (name, department, salary) VALUES (?, ?, ?)";

try (PreparedStatement preparedStatement = connection.prepareStatement(insertQuery)) {
    preparedStatement.setString(1, "John Doe");
    preparedStatement.setString(2, "IT");
    preparedStatement.setDouble(3, 75000);

    int rowsAffected = preparedStatement.executeUpdate();
    System.out.println("Rows inserted: " + rowsAffected);
}
```

---

3. Batch Updates:

Efficiently insert or update multiple records using batching.

```
String batchQuery = "INSERT INTO employees (name, department, salary) VALUES (?, ?, ?)";

try (PreparedStatement preparedStatement = connection.prepareStatement(batchQuery)) {
    connection.setAutoCommit(false); // Disable auto-commit

    // Add first batch
    preparedStatement.setString(1, "Alice");
    preparedStatement.setString(2, "Finance");
    preparedStatement.setDouble(3, 60000);
    preparedStatement.addBatch();

    // Add second batch
    preparedStatement.setString(1, "Bob");
    preparedStatement.setString(2, "HR");
```

```
    preparedStatement.setDouble(3, 55000);
    preparedStatement.addBatch();

    // Execute batch
    int[] updateCounts = preparedStatement.executeBatch();
    connection.commit(); // Commit transaction

    System.out.println("Batch executed, rows affected: " +
updateCounts.length);
}
```

---

5. Best Practices

1. **Always Close Resources**:
   - Use try-with-resources to ensure PreparedStatement and ResultSet are closed automatically.
2. **Use Parameters Appropriately**:
   - Match parameter types with the database column types to avoid conversion issues.
3. **Batch Processing**:
   - Use batching for large-scale data inserts or updates to reduce database round-trips.
4. **Avoid Hardcoding SQL**:
   - Use placeholders (?) for dynamic data instead of concatenating SQL strings.
5. **Enable Caching (Optional)**:
   - Some databases support statement caching to further optimize performance.

---

6. Comparison: Statement vs. PreparedStatement

Feature	Statement	PreparedStatement
Query	Compiles query each	Precompiled and

Feature	Statement	PreparedStatement
Execution	time.	cached.
Parameterization	Manual string concatenation.	Parameterized queries.
Security	Vulnerable to SQL injection.	Resistant to SQL injection.
Performance	Slower for repeated queries.	Faster for repeated queries.
Readability	Hard to read and maintain.	Easier to read and maintain.

By leveraging PreparedStatement, you can write efficient, secure, and maintainable database code in Java. Would you like to explore advanced use cases or discuss specific scenarios?

2. Handling database metadata

1. Handling Database Metadata in JDBC

Database metadata provides information about the database, such as its structure, capabilities, and supported features. In JDBC, the DatabaseMetaData and ResultSetMetaData interfaces are used to retrieve metadata.

---

2. DatabaseMetaData

The DatabaseMetaData interface provides information about the database as a whole, such as its capabilities, tables, schemas, and drivers.

Obtaining DatabaseMetaData

To get a DatabaseMetaData object, use the getMetaData() method of the Connection object.

Connection connection = DriverManager.getConnection(dbURL, username, password);

DatabaseMetaData databaseMetaData = connection.getMetaData();

Common Methods in DatabaseMetaData

Here are some commonly used methods:

Method	Description
getDatabaseProductName()	Returns the name of the database product.
getDatabaseProductVersion()	Returns the version of the database.
getDriverName()	Returns the name of the JDBC driver.
getTables()	Retrieves a list of tables in the database.
getColumns()	Retrieves details about table columns.
getSchemas()	Returns available schemas.

Example: Retrieving Database Information

```
try (Connection connection = DriverManager.getConnection(dbURL, username, password)) {
    DatabaseMetaData metaData = connection.getMetaData();
    System.out.println("Database Name: " + metaData.getDatabaseProductName());
    System.out.println("Database Version: " + metaData.getDatabaseProductVersion());
    System.out.println("Driver Name: " + metaData.getDriverName());


    // Retrieve tables
    ResultSet tables = metaData.getTables(null, null, "%", new String[] {"TABLE"});
    while (tables.next()) {
        System.out.println("Table: " + tables.getString("TABLE_NAME"));
    }
```

```
} catch (SQLException e) {
    e.printStackTrace();
}
```

---

3. ResultSetMetaData

The ResultSetMetaData interface provides information about the columns of a result set, such as their names, types, and sizes.

Obtaining ResultSetMetaData

To get a ResultSetMetaData object, call the getMetaData() method on a ResultSet object.

```
ResultSet resultSet = statement.executeQuery("SELECT * FROM employees");
ResultSetMetaData resultSetMetaData = resultSet.getMetaData();
```

Common Methods in ResultSetMetaData

Method	Description
getColumnCount()	Returns the number of columns in the result.
getColumnName(int column)	Returns the name of the specified column.
getColumnTypeName(int column)	Returns the SQL type of the column.
getColumnDisplaySize(int column)	Returns the display size of the column.

Example: Retrieving Column Information

```
try (Connection connection = DriverManager.getConnection(dbURL, username, password);
     Statement statement = connection.createStatement();
     ResultSet resultSet = statement.executeQuery("SELECT * FROM employees")) {
```

```
    ResultSetMetaData metaData = resultSet.getMetaData();
    int columnCount = metaData.getColumnCount();

    System.out.println("Number of Columns: " + columnCount);
    for (int i = 1; i <= columnCount; i++) {
        System.out.println("Column " + i + ": " +
metaData.getColumnName(i) +
                           " (" + metaData.getColumnTypeName(i) + ")");
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

---

4. Use Cases for Metadata

1. **Database Administration Tools**:
   - Extract database structure for documentation or visualization tools.
2. **Dynamic Query Builders**:
   - Retrieve column names and types dynamically to generate SQL queries at runtime.
3. **Validation**:
   - Validate table structures during application startup or deployment.
4. **Cross-Database Compatibility**:
   - Check supported SQL features or limits for portability across databases.

---

5. Best Practices

1. **Use Metadata Sparingly**:
   - Metadata queries can be resource-intensive. Use them judiciously to avoid performance bottlenecks.

2. **Close Resources**:
    - Always close ResultSet objects returned by metadata methods to prevent resource leaks.
3. **Database-Specific Handling**:
    - Be aware of database-specific differences in metadata (e.g., table naming conventions).
4. **Cache Results (If Applicable)**:
    - For large databases, cache metadata results if they don't change often.

By effectively leveraging DatabaseMetaData and ResultSetMetaData, you can build dynamic, adaptable, and database-aware applications. Let me know if you'd like to explore specific advanced use cases!

6. Executing stored procedures with CallableStatement

In Java, the CallableStatement interface, part of the java.sql package, is designed for executing stored procedures within a database. Stored procedures are precompiled SQL code that can perform operations such as querying or updating data. Utilizing CallableStatement allows for efficient execution of these procedures, often enhancing performance and promoting code reuse.

Steps to Execute a Stored Procedure Using CallableStatement:

1. **Establish a Database Connection:**✍
    - ✈Use the DriverManager or a DataSource to connect to your database.☛✍
2. Connection connection = DriverManager.getConnection(dbURL, username, password);
3. **Prepare the Callable Statement:**✍
    - ✈Create a CallableStatement object using the Connection.prepareCall() method.☛✍
    - ✈The SQL syntax for calling a stored procedure varies by database.☛✍

- ✈A common format is {call procedure_name(?, ?, ?)}, where each ? represents a parameter placeholder.☛✍

4. CallableStatement callableStatement = connection.prepareCall("{call procedure_name(?, ?, ?)}");
5. **Set Input Parameters (If Any):**✍
    - ✈Assign values to input parameters using the appropriate set methods, such as setInt(), setString(), etc.☛✍
6. callableStatement.setInt(1, parameterValue);
7. **Register Output Parameters (If Any):**✍
    - ✈For procedures that return output parameters, register them before execution using registerOutParameter().☛✍
    - ✈Specify the parameter index and SQL type (e.g., java.sql.Types.INTEGER).☛✍
8. callableStatement.registerOutParameter(2, java.sql.Types.INTEGER);
9. **Execute the Callable Statement:**✍
    - ✈Invoke the stored procedure using execute(), executeQuery(), or executeUpdate(), depending on the nature of the procedure.☛✍
10. callableStatement.execute();
11. **Retrieve Output Parameters (If Any):**✍
    - ✈After execution, obtain the values of output parameters using the corresponding get methods, such as getInt(), getString(), etc.☛✍
12. int result = callableStatement.getInt(2);
13. **Process Result Sets (If Any):**✍
    - ✈If the procedure returns a result set, handle it using a ResultSet object.☛✍
14. ResultSet resultSet = callableStatement.getResultSet();
15. while (resultSet.next()) {
16. // Process the result set
17. }
18. **Close Resources:**✍

- ✈Ensure that all database resources are closed to prevent resource leaks.☛✍

19. resultSet.close();
20. callableStatement.close();
21. connection.close();

Example: Calling a Stored Procedure with Input and Output Parameters

Assume a stored procedure named getEmployeeInfo that accepts an employee ID as input and returns the employee's name and salary as output.

```
try (Connection connection = DriverManager.getConnection(dbURL, username, password);
     CallableStatement callableStatement = connection.prepareCall("{call getEmployeeInfo(?, ?, ?)}")) {

    // Set input parameter
    callableStatement.setInt(1, employeeId);

    // Register output parameters
    callableStatement.registerOutParameter(2, java.sql.Types.VARCHAR);
    callableStatement.registerOutParameter(3, java.sql.Types.DECIMAL);

    // Execute the stored procedure
    callableStatement.execute();

    // Retrieve output parameters
    String employeeName = callableStatement.getString(2);
    BigDecimal employeeSalary = callableStatement.getBigDecimal(3);
```

```
    // Process the results
    System.out.println("Employee Name: " + employeeName);
    System.out.println("Employee Salary: " + employeeSalary);

} catch (SQLException e) {
    e.printStackTrace();
}
```

Key Considerations:

- **SQL Syntax Variations:** ✈The syntax for calling stored procedures can differ between database systems.☛ ✈Consult your database's documentation for the correct call syntax.☛✍
- **Error Handling:** ✈Implement robust error handling to manage SQL exceptions that may arise during database interactions.☛✍
- **Resource Management:** ✈Always close ResultSet, CallableStatement, and Connection objects in a finally block or use try-with-resources to ensure proper resource management.☛✍

✈For more detailed information, refer to the official Java documentation on CallableStatement.☛ ✂cite turn0search6☎✍

✈Additionally, practical examples and tutorials are available on platforms like GeeksforGeeks.☛ ✂cite turn0search1☎✍

✈By following these steps and considerations, you can effectively execute stored procedures in Java using CallableStatement.☛

7. Transaction Management in JDBC

Transaction management in JDBC allows you to group multiple database operations into a single unit of work. If any operation fails, the entire transaction can be rolled back, ensuring data integrity and consistency.

---

8. Key Concepts

1. **Transaction**:
    - A sequence of database operations performed as a single logical unit.

- Must adhere to the **ACID** properties:
  - **Atomicity**: All operations are completed, or none are.
  - **Consistency**: The database moves from one consistent state to another.
  - **Isolation**: Transactions do not interfere with each other.
  - **Durability**: Changes are permanent once committed.

2. **Auto-Commit Mode**:
   - By default, JDBC operates in auto-commit mode, where every SQL statement is automatically committed.
   - To manage transactions manually, auto-commit must be disabled.

---

9. Steps for Transaction Management in JDBC

1. **Disable Auto-Commit**:
   - Use setAutoCommit(false) to begin a transaction.
2. **Perform Database Operations**:
   - Execute multiple SQL statements using Statement, PreparedStatement, or CallableStatement.
3. **Commit the Transaction**:
   - Call commit() to save all changes if all operations are successful.
4. **Rollback the Transaction**:
   - Call rollback() to undo all changes if any operation fails.
5. **Enable Auto-Commit (Optional)**:
   - Re-enable auto-commit mode if necessary using setAutoCommit(true).

---

10. Example: Managing a Transaction

Here is an example where two related updates are performed in a transaction.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class TransactionManagementExample {
    public static void main(String[] args) {
        String dbURL = "jdbc:mysql://localhost:3306/mydb";
        String username = "root";
        String password = "password";

        Connection connection = null;

        try {
            // Establish connection
            connection = DriverManager.getConnection(dbURL, username, password);

            // Disable auto-commit
            connection.setAutoCommit(false);

            // Prepare SQL statements
            String updateAccount1 = "UPDATE accounts SET balance = balance - 100 WHERE id = ?";
            String updateAccount2 = "UPDATE accounts SET balance = balance + 100 WHERE id = ?";
```

```
            try (PreparedStatement stmt1 =
connection.prepareStatement(updateAccount1);
                 PreparedStatement stmt2 =
connection.prepareStatement(updateAccount2)) {

                // Execute first update
                stmt1.setInt(1, 1); // Account ID 1
                stmt1.executeUpdate();

                // Simulate an error (uncomment to test rollback)
                // if (true) throw new SQLException("Simulated error");

                // Execute second update
                stmt2.setInt(1, 2); // Account ID 2
                stmt2.executeUpdate();

                // Commit transaction
                connection.commit();
                System.out.println("Transaction committed successfully.");

            } catch (SQLException e) {
                // Rollback transaction if any error occurs
                if (connection != null) {
                    connection.rollback();
                    System.out.println("Transaction rolled back.");
                }
                e.printStackTrace();
            }

        } catch (SQLException e) {
```

```
            e.printStackTrace();
        } finally {
            // Restore auto-commit and close connection
            if (connection != null) {
                try {
                    connection.setAutoCommit(true);
                    connection.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Example 2:

```
import java.sql.*;

public class TransactionExample {

    public static void main(String[] args) {
        String url =
"jdbc:mysql://localhost:3306/your_database_name";
        String user = "your_mysql_username";
        String password = "your_mysql_password";

        try (Connection connection =
DriverManager.getConnection(url, user, password)) {

            try {
```

```
            // Disable auto-commit
            connection.setAutoCommit(false);

            // Perform database operations within the transaction
            try (Statement statement1 =
connection.createStatement();
                 Statement statement2 =
connection.createStatement()) {

                statement1.executeUpdate("UPDATE accounts SET
balance = balance - 100 WHERE id = 1");
                statement2.executeUpdate("UPDATE accounts SET
balance = balance + 100 WHERE id = 2");
            }

            // Commit the transaction
            connection.commit();
            System.out.println("Transaction committed successfully.");

        } catch (SQLException e) {
            // Rollback the transaction in case of an error
            connection.rollback();
            System.err.println("Transaction rolled back due to error: "
+ e.getMessage());
            e.printStackTrace();
        }

    } catch (SQLException e) {
        System.err.println("Connection error: " + e.getMessage());
        e.printStackTrace();
    }
```

```
    }
}
```

---

11. Key Methods for Transaction Management

1. **setAutoCommit(boolean autoCommit)**:
   - Disables/enables auto-commit mode.
   - Example: connection.setAutoCommit(false);
2. **commit()**:
   - Commits the transaction.
   - Example: connection.commit();
3. **rollback()**:
   - Rolls back the transaction.
   - Example: connection.rollback();

---

12. Best Practices

1. **Use Transactions for Critical Operations**:
   - Apply transactions to ensure atomicity for operations like transferring money or processing orders.
2. **Error Handling**:
   - Always use a try-catch block to handle SQLException and ensure proper rollback.
3. **Resource Management**:
   - Use try-with-resources to ensure Connection, Statement, and ResultSet are closed properly.
4. **Consistent State**:
   - Ensure the database returns to a consistent state if a rollback occurs.
5. **Isolation Levels**:
   - Consider setting appropriate isolation levels (e.g., READ_COMMITTED, SERIALIZABLE) to control transaction behavior.

6. connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);

---

13. Transaction Isolation Levels in JDBC

Isolation Level	Description
TRANSACTION_NONE	No transactions are supported.
TRANSACTION_READ_UNCOMMITTED	Dirty reads, non-repeatable reads, and phantom reads can occur.
TRANSACTION_READ_COMMITTED	Prevents dirty reads but allows non-repeatable reads and phantom reads.
TRANSACTION_REPEATABLE_READ	Prevents dirty and non-repeatable reads, but phantom reads can occur.
TRANSACTION_SERIALIZABLE	Prevents dirty reads, non-repeatable reads, and phantom reads.

This approach ensures robust and consistent transaction handling in JDBC applications. Would you like to explore advanced concepts like batch updates within transactions?

44. Memory Management

1. JVM Memory Structure

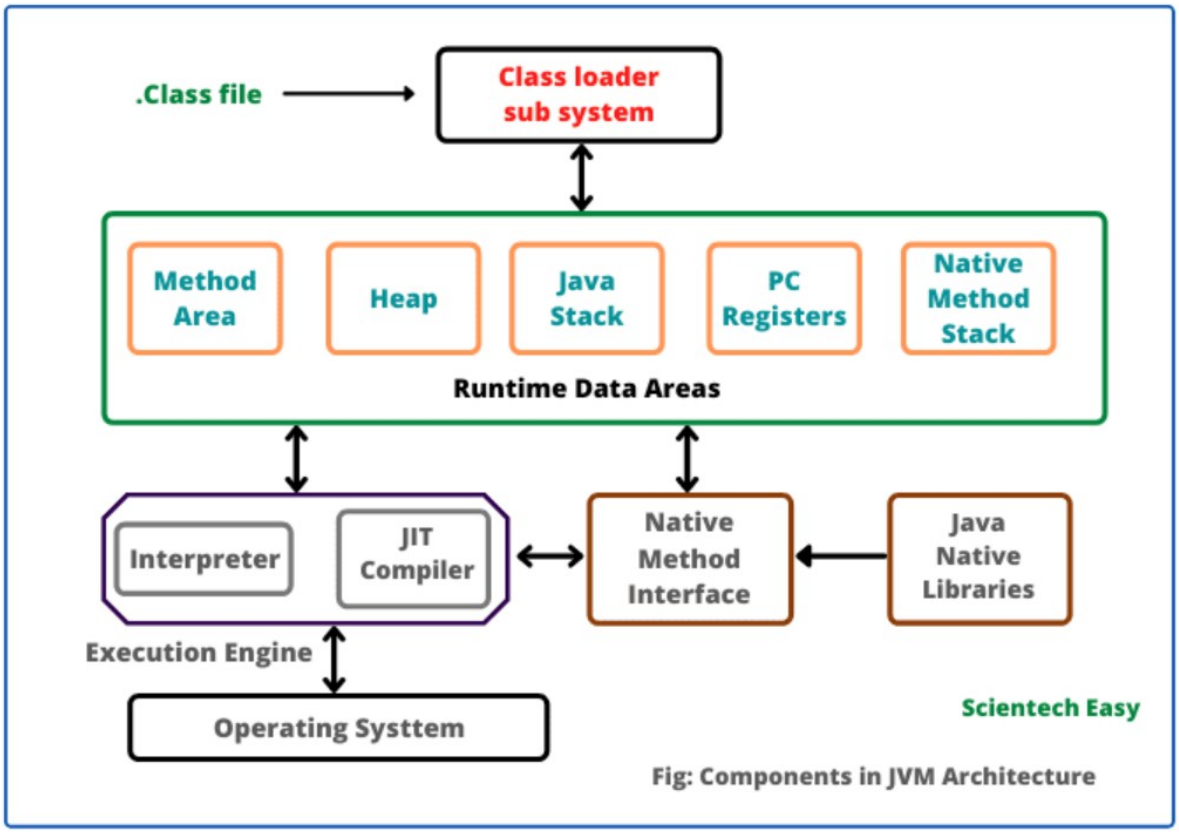


Fig: Components in JVM Architecture

1. How JVM Works Internally?

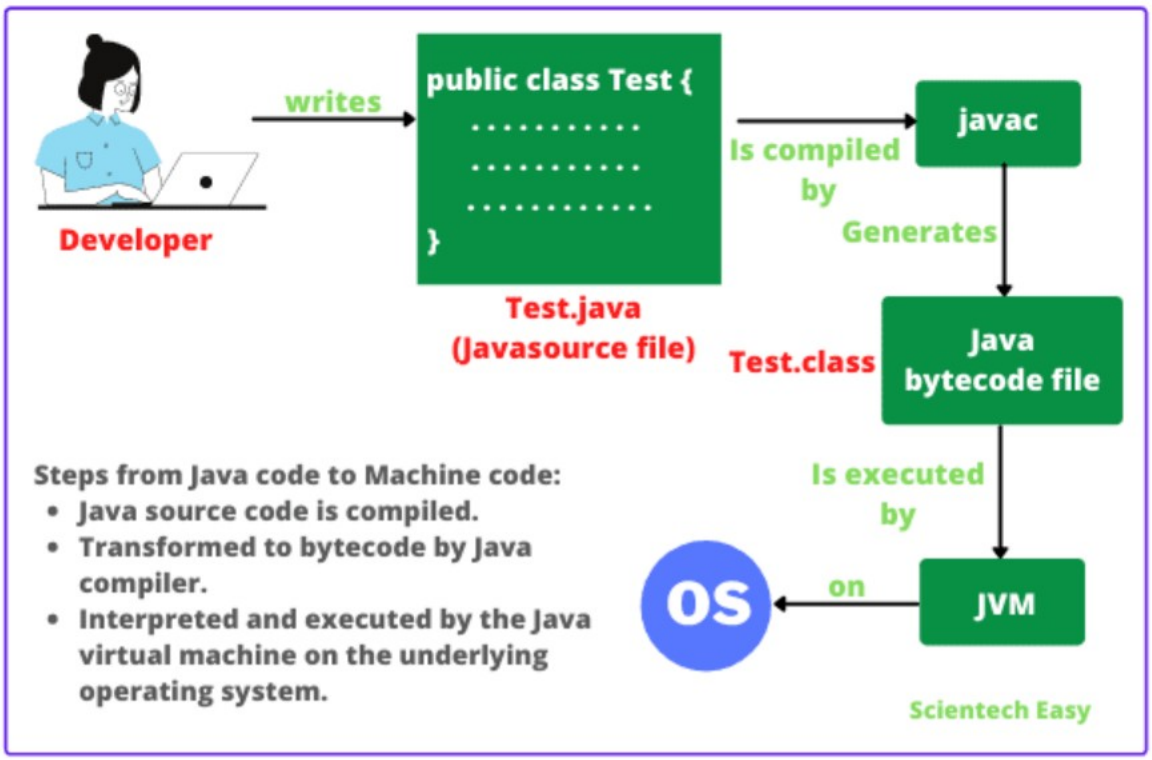
Java Virtual Machine performs the following operations for execution of the program. They are as follows:

1. Load the code into memory.
2. Verifies the code.
3. Executes the code
4. Provides runtime environment.

When we make a program in Java, .java program code is converted into a .class file consisting of byte code instructions by the Java compiler. This Java compiler is outside of JVM.

Now, Java Virtual Machine performs the following operations that are as follows:

1. This .class file is transferred to the class loader sub system of JVM as shown in the above figure.



In JVM, class loader sub system is a module or program that performs the following functions:

a) First of all, the class loader sub system loads .class file into the memory.

b) Then bytecode verifier verifies whether all byte code instructions are proper or not. If it finds any instruction suspicious, the further execution process is rejected immediately.

c) If byte code instructions are proper, it allots the necessary memory to execute the program. This memory is divided into 5 separates parts that is called **run-time data areas**. It contains the data and results during the execution of the program. These areas are as follows:

In Java, memory management is handled by the **Java Virtual Machine (JVM)**, which divides memory into distinct areas to optimize execution and manage resources efficiently. The key areas

include the **Heap**, **Stack**, and **Method Area**, each playing a unique role in the program's lifecycle.

---

1. staHeap

Definition:

The **Heap** is a portion of JVM memory used for storing objects and instance variables. When you create an object using **new**, the memory for that object is allocated from the Heap. It is shared among all threads and is the largest memory area in JVM.

Role:

- Stores objects created using the new keyword.
- Stores instance variables of objects.
- Supports garbage collection to reclaim memory occupied by unused objects.
- Shared by all threads in the JVM.

Characteristics:

- **Global Access**: Objects in the Heap are accessible from any thread.
- **Garbage Collection**: JVM automatically removes objects that are no longer referenced.
- **Divided into Regions** (in modern JVM implementations like G1 GC):
    - **Young Generation**: For newly created objects.
    - **Old Generation**: For long-lived objects.
    - **Permanent Generation** (removed in Java 8): Previously used for storing metadata.

Example:

```
class Example {
    int a;  // Stored in the Heap as part of the object instance
    String name = "Heap Memory";  // The object is stored in the Heap
}
```

---

2. Stack

Definition:

The **Stack** is a memory area used to store method call information, including local variables, parameters, and return addresses.

Role:

Each thread in a Java program has its own Stack. The Stack is used to store:

- **Method calls:** When you call a method, a new frame is pushed onto the Stack.
- **Local variables:** Variables declared within a method are stored in that method's frame on the Stack.
- **Method parameters:** Values passed to a method are also stored in the Stack frame.
- **Return addresses:** The Stack keeps track of where to return execution after a method completes.
- **Object Reference:** The reference to the object created in heap is stored in the **Stack.**

Characteristics:

- **Thread-Specific**: Each thread has its own Stack, ensuring thread safety.
- **Last In, First Out (LIFO)**: Follows the stack data structure.
- **Automatic Deallocation**: Memory for local variables is automatically deallocated when the method exits.

Stack Overflow:

Occurs stackoverflow error when the stack memory is exhausted, typically due to deep recursion or infinite method calls.

Stack Underflow:

Occurs when try to delete any element from empty stack

Example:

```
void exampleMethod() {
    int x = 10;  // Stored in the Stack
    int y = 20;  // Stored in the Stack
}
```

---

3. Method Area

Definition:

The **Method Area** is a portion of memory that stores class-level data, including metadata about classes and methods, static variables, and runtime constants.

Role:

- Stores:
  - Class-level metadata (class name, superclass, interfaces etc.).
  - Static variables.
  - Method bytecode.
  - Constant pool (e.g., string literals).
- Shared across all threads.

Characteristics:

- Part of the Heap in modern JVM implementations.
- Shared by all threads
- Read-only during execution but can be updated by JVM.
- May throw an OutOfMemoryError if it becomes full.

Example:

```
class Example {
    static int count = 0;  // Stored in the Method Area
    static void increment() {
        count++;
    }
}
```

---

4. Comparison

Feature	Heap	Stack	Method Area
Purpose	Stores objects and instance variables.	Stores method call information and local variables.	Stores class information, static variables, constants
Access	Shared across threads.	Thread-specific.	Shared across threads.
Size	Typically larger.	Smaller than Heap.	Varies depending on JVM implementation
Allocation/ Deallocation	Managed by the garbage collector.	Automatic on method entry/exit.	Managed by JVM.
Error	OutOfMemoryError if full.	StackOverflowError if full.	OutOfMemoryError if full.

5. Roles in Program Execution

1. **Heap**:
    - Stores dynamically allocated objects.
    - Memory persists as long as there is a reference to the object.
2. **Stack**:
    - Stores temporary data for method execution.

- Data is cleaned up immediately after method execution.

3. **Method Area**:
   - Stores metadata and shared data necessary for the program structure.
   - Ensures efficient reuse of static and constant data.

Memory Flow Example

Code:

```
class Example {
    static String appName = "Demo";  // Stored in Method Area
    int instanceVar = 10;           // Stored in Heap

    void display() {
        int localVar = 5;           // Stored in Stack
        System.out.println(appName + " " + instanceVar + " " + localVar);
    }
}

public class Main {
    public static void main(String[] args) {
        Example example = new Example();  // `example` reference in Stack, object in Heap
        example.display();                // `display`'s local variables in Stack
    }
}
```

Memory Breakdown:

1. appName is stored in the **Method Area** as a static variable.
2. The Example object is created in the **Heap**.

3. The reference example is stored in the **Stack** of the main thread.
4. localVar is stored in the **Stack** of the display method.

Example 2:

```
public class Example {
    static int staticVar = 10; // Stored in Method Area

    public static void main(String[] args) {
        int localVar = 5; // Stored on the Stack (main method's frame)
        Example obj = new Example(); // Object stored on the Heap
        obj.instanceMethod(20); // Method call pushes a frame onto the Stack
    }

    public void instanceMethod(int param) { // param is stored on the Stack
        int methodVar = param * 2; // Stored on the Stack (instanceMethod's frame)
    }
}
```

By separating memory into the **Heap**, **Stack**, and **Method Area**, JVM ensures efficient memory management and execution, allowing dynamic allocation, method call tracking, and class data reuse.

2. Object creation and lifecycle

1. Object Creation

2. Lifecycle of an Object

An object's lifecycle consists of the following stages:

1.Declaration

- The reference to the object is declared, but no memory is allocated.

Person person; // Declaration

2. Instantiation

- The new keyword is used to allocate memory in the Heap, and the object is created.

java

Copy code

person = new Person("Alice"); // Instantiation

3. Initialization

- A constructor is called to initialize the object.

java

Copy code

Person person = new Person("Alice"); // Initialization happens via the constructor

4. Utilization

- The object is used by invoking its methods or accessing its fields.

java

Copy code

person.greet(); // Utilization

5. Destruction

- Objects are destroyed automatically by the **Garbage Collector (GC)** when they are no longer referenced.
- The object becomes eligible for garbage collection when no live thread can access it.

person = null; // Makes the object eligible for GC

3. Garbage Collection

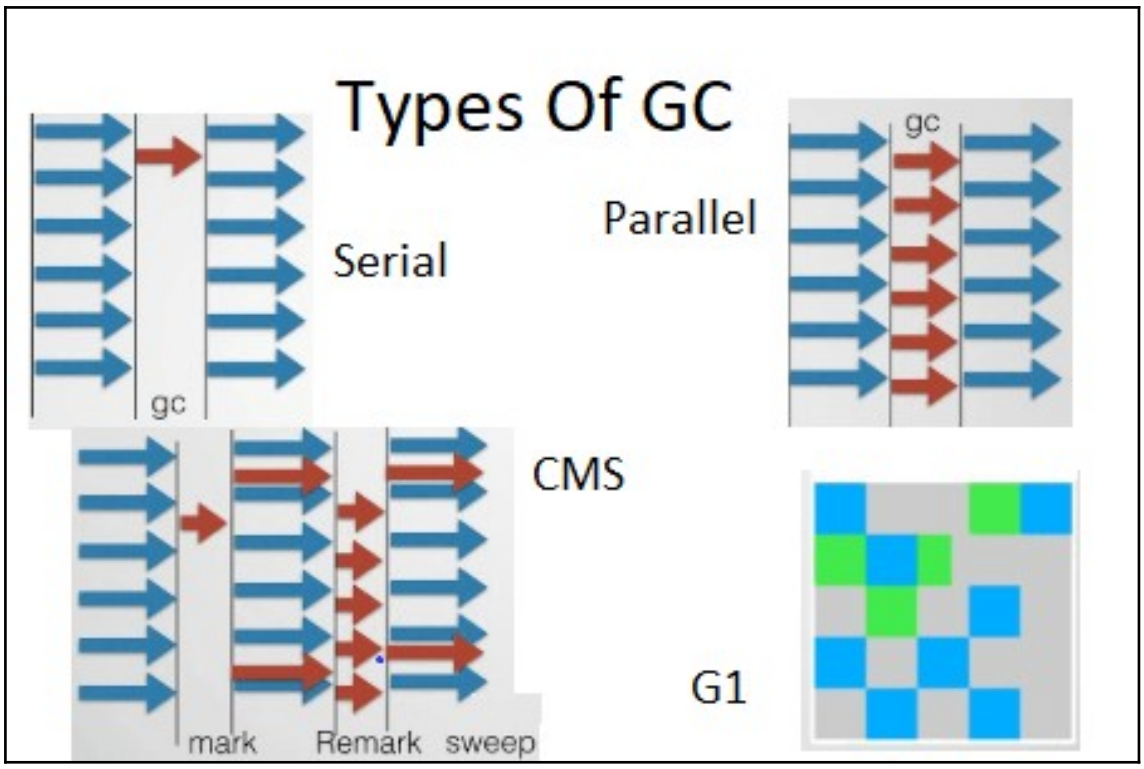
Java's **Garbage Collection (GC)** process automatically reclaims memory occupied by objects that are no longer accessible, ensuring efficient memory management. Several GC algorithms are implemented in the JVM, each optimized for specific scenarios, offering trade-offs in terms of throughput, latency, and memory usage.

1. Key Concepts Before We Start

- **Stop-the-World (STW):** A phase during GC where all application threads are **paused**. This is necessary for some GC operations to ensure data consistency. Minimizing STW pauses is a major goal of modern GC algorithms.
- **Generational GC:** Most modern GCs use a generational approach, dividing the heap into generations (Young Generation, Old Generation). Objects are typically created in the Young Generation. Objects that survive multiple GC cycles in the Young Generation are moved to the Old Generation. This is based on the observation that most objects have short lifespans.
- **Minor GC:** GC that occurs in the Young Generation.
- **Major GC (or Full GC):** GC that occurs in the Old Generation (and potentially other areas of the heap). Major GCs typically involve longer STW pauses.

2. Types of GC algorithms

1. Serial Garbage Collector
2. Parallel Garbage Collector
3. CMS (Concurrent Mark Sweep)
4. G1 Garbage Collector
5. Z Garbage Collector



3. Serial Garbage Collector

4. **Overview**:
   - A simple, single-threaded garbage collector.
   - Best suited for applications with small heaps and single-threaded environments.

5. How It Works:

- Performs **Stop-The-World (STW)** pauses to collect garbage.
- Utilizes **mark-and-sweep** and **compact** phases sequentially.

6. Advantages:

- Low overhead due to single-threaded operation.
- Simple implementation.

7. Disadvantages:

- Not suitable for multi-threaded applications or large heaps.

- Long pause times as the entire application is stopped during GC.

8. Use Case:

- Single-threaded applications with small heaps (e.g., desktop apps).

9. Enable:

bash

Copy code

-XX:+UseSerialGC

---

4. Parallel Garbage Collector

5. Overview:

- Also called the "Throughput Collector."
- Focuses on maximizing throughput by using multiple threads for GC operations.

6. How It Works:

- Executes **young generation** collections and compaction in parallel using multiple threads.
- Performs **Stop-The-World** pauses for old generation collections.

7. Advantages:

- Efficient for multi-threaded applications.
- High throughput and short GC times.

8. Disadvantages:

- Long pause times for large heaps.
- Does not prioritize low latency.

9. Use Case:

- Batch-processing systems or applications prioritizing throughput over low latency.

10. Enable:

-XX:+UseParallelGC

---

5. Concurrent Mark-Sweep (CMS) Collector

6. Overview:

- Focuses on reducing pause times by performing most of its GC work concurrently with the application.

7. How It Works:

- **Young generation**: Uses a parallel collector.
- **Old generation**: Performs a **mark-and-sweep** algorithm with concurrent marking and sweeping phases.
- Minimal **Stop-The-World** pauses are required for initial marking and remarking.

8. Advantages:

- Low pause times, making it ideal for latency-sensitive applications.
- Performs most GC work concurrently.

9. Disadvantages:

- Higher CPU overhead due to concurrent threads.
- Does not compact memory, leading to potential fragmentation.

10. Use Case:

- Low-latency applications (e.g., web servers, interactive applications).

11. Enable:

bash

Copy code

-XX:+UseConcMarkSweepGC

---

6. G1 Garbage Collector

7. Overview:

- Replaces CMS in modern JVMs.

- Balances throughput and latency by dividing the heap into **regions**.

8. How It Works:

- Divides the heap into equal-sized regions.
- Performs garbage collection incrementally:
  - **Young Generation**: Uses parallel threads.
  - **Old Generation**: Collects only selected regions based on priority ("garbage-first").
- Supports concurrent compaction to minimize fragmentation.

9. Advantages:

- Predictable pause times via configurable MaxGCPauseMillis.
- Incremental compaction reduces fragmentation.

10. Disadvantages:

- Complex tuning for specific workloads.
- Slightly higher CPU usage compared to CMS.

11. Use Case:

- Applications requiring both low latency and moderate throughput (e.g., large-scale systems).

12. Enable:

bash

Copy code

-XX:+UseG1GC

---

7. Z Garbage Collector (ZGC)

8. Overview:

- Designed for ultra-low-latency applications.
- Handles large heaps (up to terabytes) with minimal pause times.

9. How It Works:

- Utilizes a **region-based** approach.

- Performs almost all GC operations concurrently with application threads.
- Pause times are typically in the range of a few milliseconds, regardless of heap size.

10. **Advantages:**
    - Extremely low pause times.
    - Efficient with large heaps and highly concurrent applications.
11. **Disadvantages:**
    - Higher memory overhead (requires additional memory for concurrent processing).
    - Supported only in newer JVM versions.
12. **Use Case:**
    - Latency-critical systems (e.g., financial systems, gaming, real-time analytics).
13. **Enable:**

bash

Copy code

-XX:+UseZGC

---

8. Comparison of GC Algorithms

GC Algorithm	Key Focus	Heap Size	Pause Time	Throughput	Use Case
Serial GC	Simplicity	Small	High	Moderate	Single-threaded, small applications
Parallel GC	Throughput	Medium to Large	High	High	Batch jobs, throughput-focused apps
CMS GC	Low latency	Medium to Large	Low	Moderate	Low-latency, interactive systems

GC Algorithm	Key Focus	Heap Size	Pause Time	Throughput	Use Case
G1 GC	Balanced	Medium to Large	Configurable	High	Mixed workloads
ZGC	Ultra-low latency	Large (up to TBs)	Minimal	Moderate	Real-time, latency-critical apps

9. Key Terms

1. **Stop-The-World (STW)**:
   - Application threads are paused during GC.
   - Goal is to minimize STW events for better performance.
2. **Throughput**:
   - The percentage of time spent running the application versus performing GC.
3. **Latency**:
   - The time an application is paused for garbage collection.

---

10. Choosing the Right GC Algorithm

- **Small Apps**: Use **Serial GC** for simplicity.
- **Throughput-Intensive Apps**: Use **Parallel GC**.
- **Low Latency Apps**: Use **CMS GC** or **G1 GC**.
- **Large Heap Apps**: Use **G1 GC** or **ZGC**.
- **Ultra-Low Latency**: Use **ZGC** for critical real-time systems.

4. Tuning and optimizing garbage collection

Tuning the garbage collection (GC) process in Java is crucial for achieving optimal application performance, especially in memory-intensive or latency-sensitive applications. Proper GC tuning ensures minimal pause times, balanced throughput, and efficient memory utilization.

---

1. Understand Application Requirements

Before tuning, determine:

- **Throughput Priority**: Maximize application processing time.
- **Latency Priority**: Minimize pause times.
- **Memory Constraints**: Optimize for limited heap space.

---

2. Key Garbage Collection Metrics

- **GC Pause Time**: Duration for which the application is paused during garbage collection.
- **Throughput**: Percentage of time the application spends executing rather than performing GC.
- **Heap Usage**: Memory used by the application and GC at any given time.
- **Frequency of Collections**: Number of GC events over a specific time period.

---

3. Steps for Tuning Garbage Collection

4. Step 1: Choose the Right Garbage Collector

- Use the most appropriate GC for your application needs:
    - **Serial GC**: For small, single-threaded applications.
    - -XX:+UseSerialGC
    - **Parallel GC**: For high throughput applications.
    - -XX:+UseParallelGC
    - **G1 GC**: For balanced throughput and latency.
    - -XX:+UseG1GC
    - **ZGC**: For ultra-low-latency applications.
    - -XX:+UseZGC

5. Step 2: Configure Heap Sizes

Set appropriate heap sizes to prevent frequent GC events:

- **Initial Heap Size**:
- -Xms<size>

- **Maximum Heap Size**:
- -Xmx<size>

For example:

-Xms512m -Xmx4g

6. Step 3: Adjust Generation Sizes

- **Young Generation**: Frequent, short-lived objects.
- **Old Generation**: Long-lived objects.

Control their sizes:

-XX:NewRatio=<ratio> # Ratio of old generation to young generation.

-XX:NewSize=<size> # Initial young generation size.

-XX:MaxNewSize=<size> # Maximum young generation size.

7. Step 4: Set Pause Time Goals

- For **G1 GC**, set desired pause time:
- -XX:MaxGCPauseMillis=<time_in_ms>

8. Step 5: Tune Parallelism

- Control the number of threads for GC operations:
- -XX:ParallelGCThreads=<num_threads>
- -XX:ConcGCThreads=<num_threads>

9. Step 6: Enable GC Logging

- Use GC logs to monitor and analyze garbage collection behavior:
- -Xlog:gc*:file=gc.log:time,uptime,level,tags

Example output:

[0.153s][info][gc,start] GC(0) Pause Young (G1 Evacuation Pause)

[0.154s][info][gc,end] GC(0) Pause Young (G1 Evacuation Pause) 8M->4M(32M) 1ms

10. Step 7: Enable Heap Dumps

- Generate heap dumps to analyze memory usage:
- -XX:+HeapDumpOnOutOfMemoryError
- -XX:HeapDumpPath=<file_path>

11. **Step 8: Optimize Object Allocation**
    - Minimize object creation and reuse objects where possible.
    - Use **StringBuilder** instead of String for concatenation.
12. **Step 9: Use Efficient Data Structures**
    - Replace HashMap with ConcurrentHashMap if needed.
    - Use primitive arrays instead of boxed types (e.g., int[] vs. Integer[]).

---

4. Common Tuning Scenarios

5. High Throughput Applications

- Use **Parallel GC**.
- Maximize young generation size to reduce old generation collections.
- Example:
- -XX:+UseParallelGC -Xms2g -Xmx4g -XX:NewRatio=2

6. Low-Latency Applications

- Use **G1 GC** or **ZGC**.
- Focus on reducing pause times:
- -XX:+UseG1GC -XX:MaxGCPauseMillis=100

7. Memory-Constrained Applications

- Use **CMS GC** or **G1 GC** to manage memory efficiently.
- Monitor and limit heap size.
- Example:
- -XX:+UseConcMarkSweepGC -Xms512m -Xmx1g

---

5. Tools for GC Analysis

6. VisualVM

- A free, graphical monitoring tool.
- Displays heap usage, GC events, and thread activity.

7. Eclipse MAT

- Analyzes heap dumps for memory leaks and object retention.

8. JConsole

- Monitors JVM memory, thread usage, and GC activity in real-time.

9. GCViewer

- Visualizes GC logs for better understanding of GC behavior.

---

6. Practical Example

7. Command-Line Options

java -Xms512m -Xmx4g -XX:+UseG1GC -XX:MaxGCPauseMillis=200 -Xlog:gc*:gc.log MyApp

8. Sample Output Analysis

From the GC log:

[5.123s][info][gc] GC(45) Pause Young (Normal) 512M->256M(4G) 12ms

[15.456s][info][gc] GC(46) Pause Young (Mixed) 256M->128M(4G) 30ms

- **Heap Usage**: Reduced from 512M to 256M.
- **Pause Time**: GC completed in 12ms, meeting the pause time goal.

---

7. Key Considerations

1. **Avoid Over-Tuning**:
    - Start with defaults and only adjust parameters when necessary.
2. **Monitor Performance**:
    - Use tools to continuously observe application behavior under real-world loads.
3. **Test Changes**:
    - Validate tuning changes in a staging environment before deploying.

By combining the right GC algorithm, heap configuration, and analysis tools, you can optimize Java's garbage collection to meet your application's performance goals.

5. Memory Leak Identification and Handling

1. Common causes and symptoms of memory leaks

A memory leak occurs when objects that are no longer needed by an application remain in memory and cannot be reclaimed by the Garbage Collector (GC). While Java's GC handles most memory management automatically, certain programming patterns can still lead to leaks.

---

6. Common Causes of Memory Leaks

1. **Unclosed Resources**:
   - Failure to close streams, connections, or other resources.

   Example:
   - FileInputStream fis = new FileInputStream("file.txt");
   - // If not closed, it can cause a memory leak

   Solution: Use try-with-resources.
   - try (FileInputStream fis = new FileInputStream("file.txt")) {
   - // Use the resource
   - }
2. **Static References**:
   - Objects referenced by static fields are retained for the lifetime of the class.
   - **Example**:
   - class Example {
   - static List<String> cache = new ArrayList<>();
   - }
   - **Solution**: Use weak references or explicitly clear static references when no longer needed.

3. **Listener or Callback References**:
    - Event listeners or callbacks registered but not unregistered when no longer needed.
    - **Example**:
    - button.addActionListener(listener);
    - **Solution**: Unregister listeners when they are no longer needed.
4. **Inner Classes and Anonymous Classes**:
    - Implicit references to the outer class instance prevent garbage collection.

    Example:
    - class Outer {
    - class Inner {
    - void doSomething() { }
    - }
    - }
    - **Solution**: Use static inner classes or avoid capturing the outer class unnecessarily.
5. **Collections with References**:
    - Objects stored in collections (e.g., HashMap, ArrayList) are not removed properly.

    Example:
    - Map<Key, Value> cache = new HashMap<>();
    - // If 'Value' objects are no longer needed but remain in the map

    Solution: Remove unused objects from collections or use weak collections like WeakHashMap.
6. **ThreadLocal Variables**:
    - Values stored in ThreadLocal can leak memory if not explicitly removed.
    - **Solution**: Use remove() to clear values when the thread is done.

- threadLocal.remove();

7. **Custom ClassLoaders**:
   - Improperly managed class loaders can prevent classes and objects from being garbage collected.
   - Common in web applications where class loaders are reused.
8. **Large Object Graphs**:
   - Complex object graphs with circular references can cause issues if references are not nullified.
9. **Caching Without Eviction Policies:**
   - Caches are used to improve performance by storing frequently accessed data in memory. However, if a cache does not have an eviction policy (e.g., Least Recently Used (LRU), time-based eviction), it can grow indefinitely and lead to a memory leak

---

2. Symptoms of Memory Leaks

1. **Increased Memory Usage Over Time**:
   - The application consumes more memory as it runs, eventually leading to an OutOfMemoryError.
2. **Frequent Garbage Collection**:
   - Increased GC activity, with little improvement in memory usage, indicates objects cannot be reclaimed.
3. **Performance Degradation**:
   - Slower response times due to increased GC activity and reduced available heap memory.
4. **OutOfMemoryError**:
   - The JVM throws an error when it cannot allocate memory for new objects.
     - Example:
     - Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
5. **Heap Dumps**:

- Analysis of heap dumps reveals a growing number of unreachable objects still retained in memory.

6. **Application Crashes**:
   - Memory leaks can eventually exhaust available memory, causing the application to terminate unexpectedly.

---

3. Detecting and Resolving Memory Leaks

4. Tools for Detection

1. **VisualVM**:
   - Analyze heap memory, thread activity, and GC performance.
2. **Eclipse MAT (Memory Analyzer Tool)**:
   - Analyze heap dumps to identify retained objects and leaks.
3. **JProfiler**:
   - Provides memory profiling and leak detection.
4. **YourKit Java Profiler**:
   - Offers detailed memory usage analysis and leak detection.
5. **GC Logs**:
   - Enable GC logging to monitor memory usage patterns.

5. Best Practices for Prevention

1. **Use Modern APIs**:
   - Utilize try-with-resources for proper resource management.
2. **Avoid Static References**:
   - Only use static fields when absolutely necessary.
3. **Explicit Cleanup**:
   - Remove objects from collections and unregister listeners or callbacks.
4. **Weak References**:

- Use WeakHashMap or WeakReference for objects that should not prevent GC.

5. **Memory Profiling**:
   - Regularly profile the application during development and production to catch leaks early.
6. **Test for Leaks**:
   - Use tools like JUnit with memory assertions to verify that objects are properly reclaimed.

---

6. Example of a Memory Leak

```
import java.util.ArrayList;
import java.util.List;

public class MemoryLeakExample {
    private static List<byte[]> leak = new ArrayList<>();

    public static void main(String[] args) {
        while (true) {
            byte[] b = new byte[1024 * 1024]; // Allocate 1MB
            leak.add(b); // Retain the object in the static list
            System.out.println("Allocated memory: " + leak.size() + "
MB");
        }
    }
}
```

Symptoms:

- Heap memory usage will continuously increase.
- Eventually, the program will throw an OutOfMemoryError.

7. Fix:

- Avoid retaining objects unnecessarily:

- leak.clear();

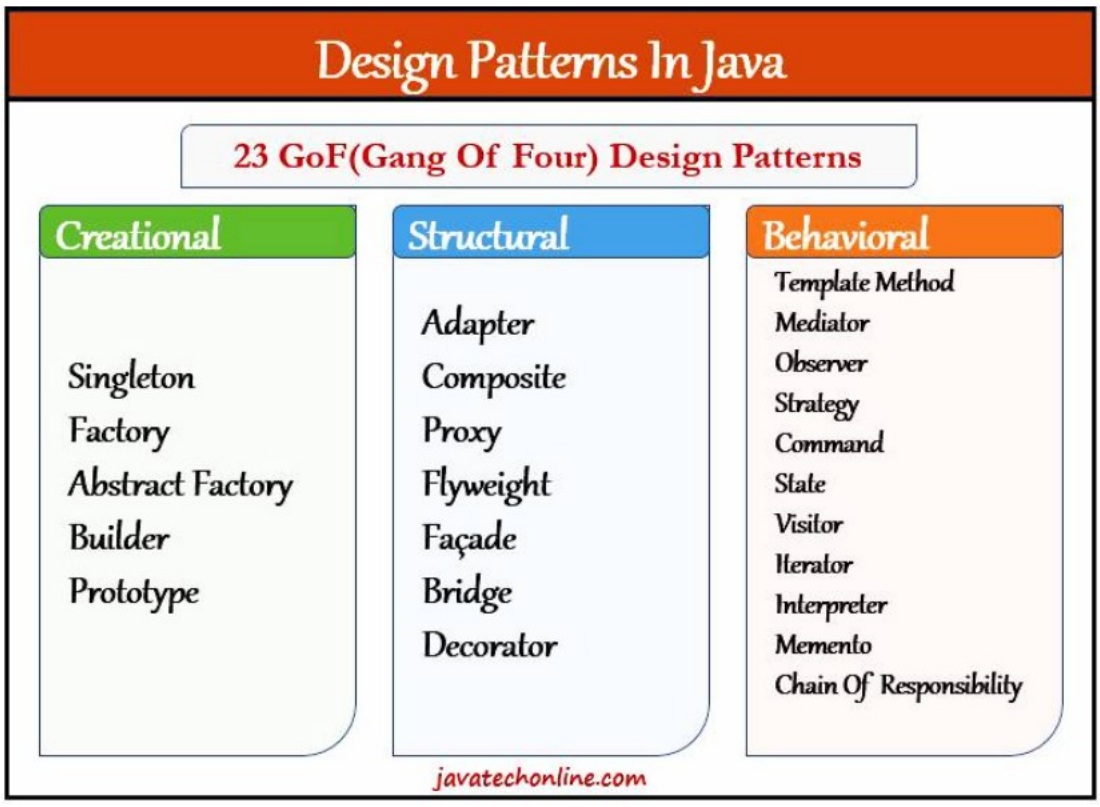
By understanding the causes, symptoms, and tools for detecting memory leaks, developers can ensure better resource management and prevent performance issues in Java applications.

45. Desing Pattern

Design patterns in Java are standardized solutions to common problems in software design. They represent best practices and are widely used to create robust, reusable, and maintainable code.

Type of Design Pattern

1. Creational
2. Structural
3. Behavioural

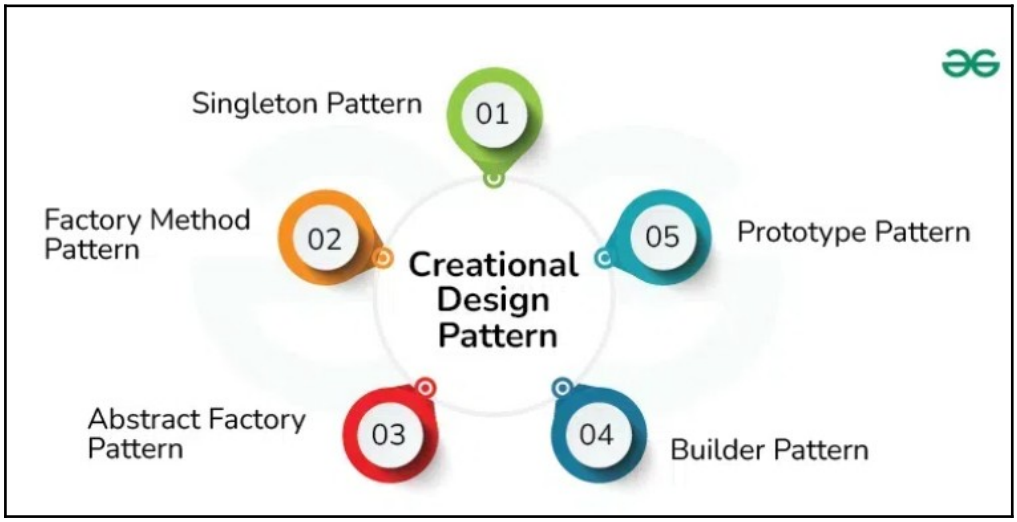


The term **"Gang of Four" (GoF)** refers to the four authors of the book *Design Patterns: Elements of Reusable Object-Oriented Software*, which was published in 1994. The authors are:

1. **Erich Gamma**
2. **Richard Helm**
3. **Ralph Johnson**
4. **John Vlissides**

1. Creational Desing Pattern

These patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation



1. Singleton Pattern

The Singleton Pattern in Java is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. It is commonly used when exactly one instance of a class is needed to coordinate actions across the system

Key Characteristics of Singleton Pattern

1. **Private Constructor**: Prevents instantiation of the class from outside.

2. **Static Instance**: A static variable holds the single instance of the class.
3. **Global Access Point**: A public static method provides the single instance to clients.

Ways to impleemnt Singleton Class

1. Eager Initialization
2. Lazy Initialization
3. Thread-Safe Singleton
4. Double-Checked Locking (Thread-Safe and Efficient)
5. Bill Pugh Singleton (Using Static Inner Class)

1. Eager Initilization:

This is the simplest method of creating a singleton class. In this, object of class is created when it is loaded to the memory by JVM. It is done by assigning the reference of an instance directly.

Pros:

1. Very simple to implement.
2. May lead to resource wastage. Because instance of class is created always, whether it is required or not.
3. CPU time is also wasted in creation of instance if it is not required.
4. Exception handling is not possible.

2. Lazy Initialization:

In this method, object is created only if it is needed. This may prevent resource wastage. An implementation of getInstance() method is required which return the instance. There is a null check that if object is not created then create, otherwise return previously created. To make sure that class cannot be instantiated in any other way, constructor is made private. As object is created with in a method, it ensures that object will not be created until and unless it is required. Instance is kept private so that no one can access it directly.

it can be used in a single threaded environment because multiple threads can break singleton property as they can access get instance method simultaneously and create multiple objects.

Pros:

1. Object is created only if it is needed. It may overcome wastage of resource and CPU time.
2. Exception handling is also possible in method.
3. Every time a condition of null has to be checked.
4. instance can't be accessed directly.
5. In multithreaded environment, it may break singleton property.

3. Thread Safe Singleton:

A thread safe singleton is created so that singleton property is maintained even in multithreaded environment. To make a singleton class thread safe, getInstance() method is made synchronized so that multiple threads can't access it simultaneously.

Pros:

1. Lazy initialization is possible.
2. It is also thread safe.
3. getInstance() method is synchronized so it causes slow performance as multiple threads can't access it simultaneously.

4. Double-Checked Locking (Thread-Safe and Efficient):

In this mechanism, we overcome the overhead problem of synchronized code. In this method, getInstance is not synchronized but the block which creates instance is synchronized so that minimum number of threads have to wait and that's only for first time.

Pros:

1. Lazy initialization is possible.
2. It is also thread safe.
3. Performance overhead gets reduced because of synchronized keyword.

4. First time, it can affect performance.

5. Bill Pugh Singleton (Using Static Inner Class):

Prior to Java5, memory model had a lot of issues and above methods caused failure in certain scenarios in multithreaded environment. So, Bill Pugh suggested a concept of inner static classes to use for singleton.

When the singleton class is loaded, inner class is not loaded and hence doesn't create object when loading the class. Inner class is created only when getInstance() method is called. So it may seem like eager initialization but it is lazy initialization.
This is the most widely used approach as it doesn't use synchronization.

When to use What

1. Eager initialization is easy to implement but it may cause resource and CPU time wastage. Use it only if cost of initializing a class is less in terms of resources or your program will always need the instance of class.
2. By using Static block in Eager initialization we can provide exception handling and also can control over instance.
3. Using synchronized we can create singleton class in multi-threading environment also but it can cause slow performance, so we can use Double check locking mechanism.
4. Bill Pugh implementation is most widely used approach for singleton classes. Most developers prefer it because of its simplicity and advantages.

Example Use Case

Singletons are useful for managing shared resources, such as:

- Database connection pools
- Configuration settings
- Logging frameworks
- Cache management

Advantages of Singleton Pattern

- Ensures control over a single instance.

- Saves memory by avoiding multiple object creations.
- Provides a global access point.

Disadvantages of Singleton Pattern

- Harder to test due to global state.
- Can introduce tight coupling if not used properly.

Problems if Singleton Design Pattern is not implemented

The Singleton design pattern is used to ensure that a class has only one instance and provides a global point of access to it. While it can be useful in certain situations, failing to follow the Singleton pattern when it's appropriate can lead to several problems:

1. **Uncontrolled Instantiation:** Without a Singleton, multiple instances of a class can be created. This can lead to:

- **Wasted Resources:** Each instance consumes memory and other resources. Multiple instances might waste resources unnecessarily, especially if the class manages expensive resources like database connections or file handles.

- **Inconsistent State:** If multiple instances manage shared data, changes made by one instance might not be reflected in others, leading to inconsistencies and unexpected behavior.

- **Incorrect Program Logic:** Some classes are designed with the assumption that only one instance exists. Creating multiple instances can break the program's logic and lead to errors.

2. **Difficulty in Managing Shared Resources:** Singletons are often used to manage access to shared resources. Without a Singleton, it becomes difficult to coordinate access to these resources, potentially leading to:

- **Race Conditions:** Multiple instances might try to access and modify the shared resource concurrently, leading to race conditions and data corruption.

- **Deadlocks:** If instances acquire locks on resources in different orders, it can lead to deadlocks, where instances are blocked indefinitely waiting for each other.

3. **Tight Coupling and Reduced Testability:** While Singletons themselves can introduce tight coupling, not using them when appropriate can also lead to similar problems:

- **Global State:** If multiple instances are used to mimic a Singleton's behavior, it can effectively create global state, making it difficult to reason about the program's behavior and making it harder to test individual components in isolation.
- **Increased Complexity:** Managing multiple instances and their interactions can add complexity to the code, making it harder to understand and maintain.

Example:

Consider a logging class that writes messages to a file. If this class is not implemented as a Singleton, multiple parts of the application might create their own instances. This could lead to:

- Multiple log files being created.
- Log messages being interleaved in unexpected ways, making it difficult to debug issues.
- Resource contention if multiple instances try to write to the same file simultaneously.

By using a Singleton for the logging class, you can ensure that all log messages are written to a single file in a consistent order, simplifying debugging and resource management.

In conclusion: While the Singleton pattern should be used judiciously, failing to use it when appropriate can lead to various problems, including wasted resources, inconsistent state, difficulty in managing shared resources, and reduced testability.

2. Factory Design Pattern

The Factory Method is a creational design pattern that defines an interface or abstract class for creating objects, but lets subclasses decide which class to instantiate. In other words, it defers the instantiation logic to subclasses.

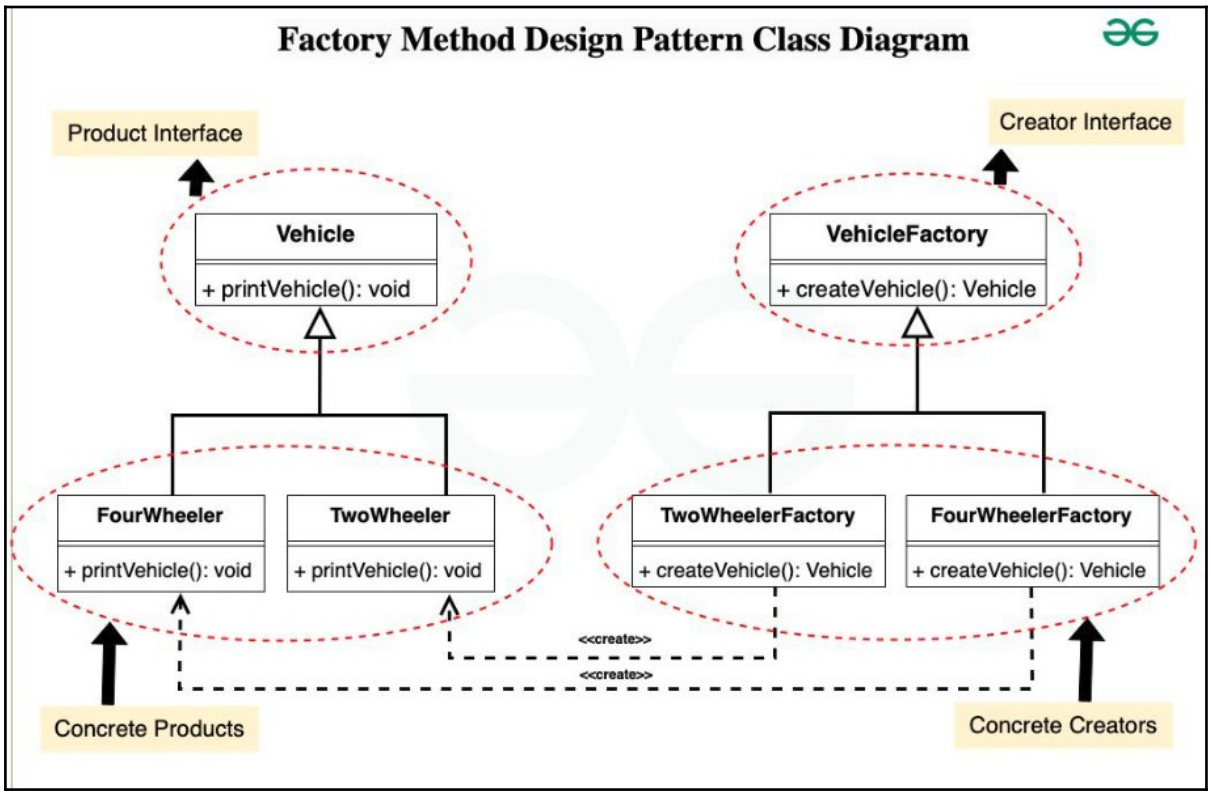
Problem:

Imagine you have a class that needs to create objects of various types. If the logic for creating these objects is directly within the class, it becomes tightly coupled to those specific object types. This makes it difficult to add new object types or modify existing ones without changing the original class.

Solution:

The Factory Method pattern solves this by:

- Defining a factory interface or abstract class with a factory method. This method returns an object of a certain type (the product).
- Creating concrete factory subclasses that implement the factory method and return concrete product objects.



Structure:

- **Product:** Defines the interface of objects the factory method creates.

- **Concrete Product:** Concrete implementations of the Product interface.
- **Creator:** Declares the factory method, which returns an object of type Product. The Creator may also define a default implementation of the factory [1] method that returns a default Concrete Product object.
- **Concrete Creator:** Overrides the factory method to return an instance of a Concrete Product.

1. **Tight Coupling:** The WithoutFactoryPattern class is directly dependent on the concrete Circle, Square, and Rectangle classes. If you need to add a new shape (e.g., Triangle), you would have to modify the WithoutFactoryPattern class, violating the Open/Closed Principle (open for extension, closed for modification).
2. **Creation Logic Scattered:** If the creation of objects becomes more complex (e.g., needing to pass parameters to constructors, performing initialization), this logic gets scattered throughout the client code, making it harder to maintain and understand. In the example, we simulate this with the shapeType input. Imagine if the Circle required a radius in its constructor. Every place you create a Circle would need to handle that logic.
3. **Code Duplication:** If you need to create shapes in multiple parts of your application, you'll end up duplicating the creation logic, leading to code bloat and increased maintenance effort.
4. **Difficult to Change Creation Logic:** If you decide to change *how* shapes are created (e.g., using a different constructor, using a pool of objects), you would have to modify every place where shapes are created.

How the Factory Pattern Solves These Issues:

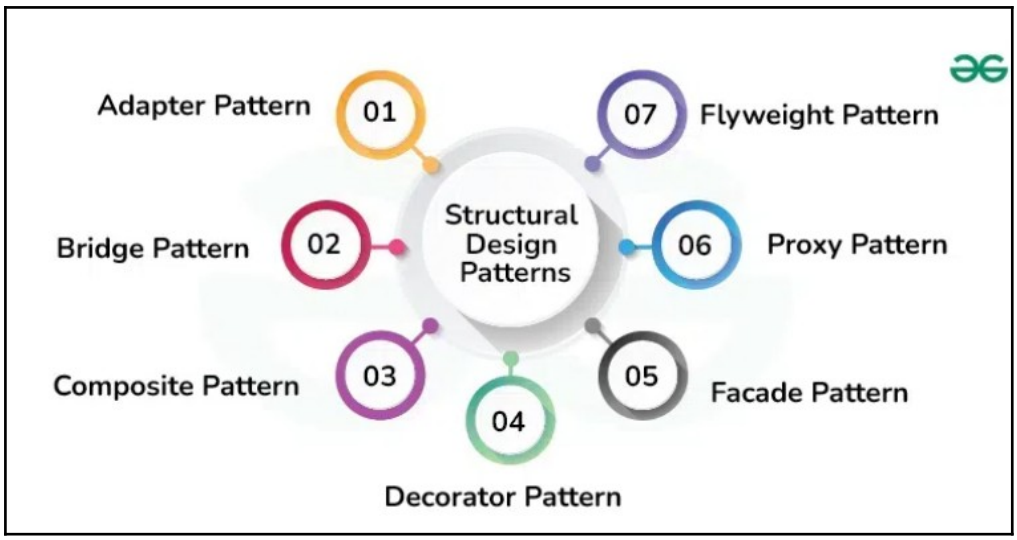
1. **Decoupling:** The WithFactoryPattern client code interacts with the ShapeFactory interface, not the concrete shape classes directly. This decouples the client from the concrete implementations.
2. **Centralized Creation Logic:** The creation logic is encapsulated within the concrete factory classes

(CircleFactory, SquareFactory, etc.). This makes it easier to manage and change the creation process.

3. **Open/Closed Principle:** You can add new shape types by creating new concrete Shape classes and corresponding ShapeFactory classes *without* modifying existing client code.
4. **Code Reusability:** The factory classes can be reused throughout the application, eliminating code duplication.

2. Structural Desing Pattern

These patterns compose classes or objects into larger structures while keeping the structures flexible and efficient



1. Adaptor Desing Pattern:

The Adapter design pattern is a structural pattern that allows objects with incompatible interfaces to work together. It [1] acts as a bridge between two incompatible interfaces, converting the interface of one class [2] into an interface that the client expects.

Problem:

Imagine you have a client that expects to interact with objects through a specific interface (let's call it the "Target" interface). However, you have a class (the "Adaptee") that provides the required functionality but has a different interface. You can't directly use the Adaptee with the client because of this interface mismatch.

Solution:

The Adapter pattern solves this problem by introducing an "Adapter" class. The Adapter:

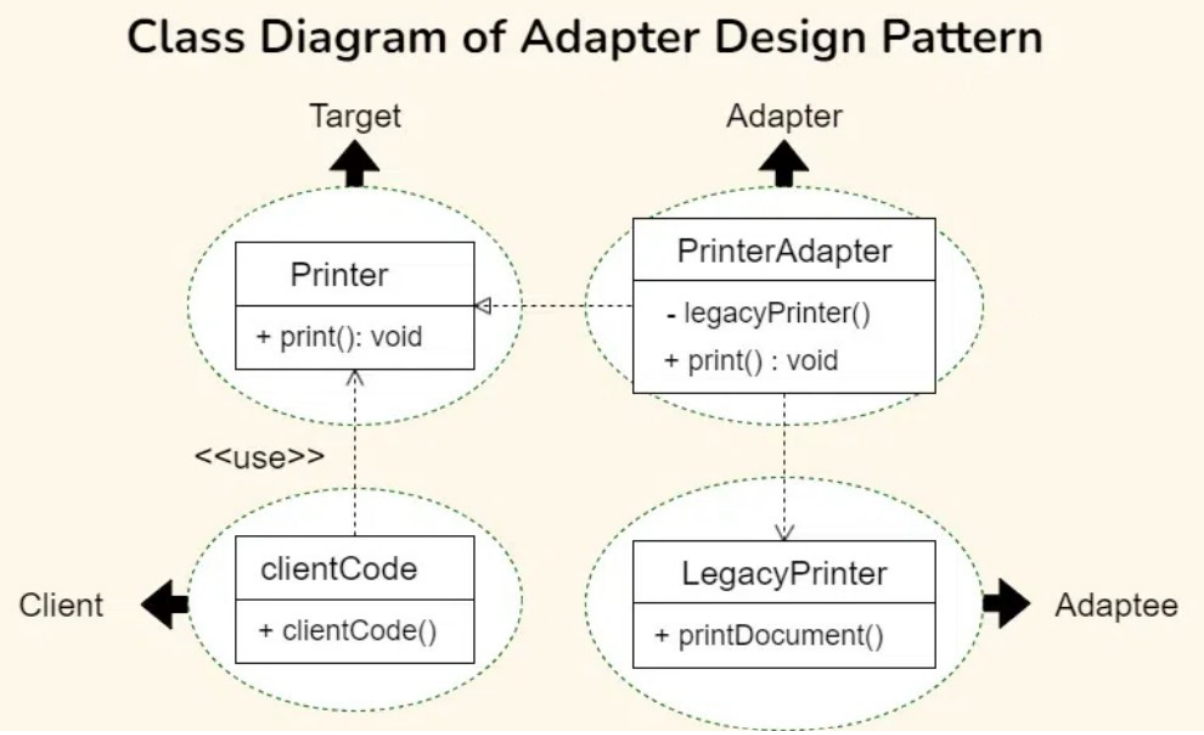
- Implements the Target interface (the interface the client expects).
- Holds an instance of the Adaptee class.
- Delegates calls from the client to the Adaptee, translating the calls as necessary to match the Adaptee's interface.

Types of Adapters:

1. **Object Adapter (Using Composition):** The Adapter holds an instance of the Adaptee. This is the more common and recommended approach.
2. **Class Adapter (Using Inheritance):** The Adapter inherits from both the Target and the Adaptee. This approach is less flexible because Java doesn't support multiple inheritance of classes.

Structure (Object Adapter):

- **Target Interface:** Defines the interface that the client uses.
- **Adaptee:** The existing class with an incompatible interface.
- **Adapter:** Implements the Target interface and holds an instance of the Adaptee. It translates calls from the Target interface to the Adaptee's interface.
- **Client:** Uses the Target interface to interact with objects.



Example (Converting Celsius to Fahrenheit):

Let's say you have a client that works with Fahrenheit temperatures, but you have a temperature sensor that provides readings in Celsius.

Java

```
// Target Interface (Fahrenheit)
interface FahrenheitTemperature {
    double getFahrenheit();
}

// Adaptee (Celsius)
class CelsiusTemperature {
    private double celsius;

    public CelsiusTemperature(double celsius) {
        this.celsius = celsius;
    }

    public double getCelsius() {
        return celsius;
    }
}

// Adapter (Celsius to Fahrenheit)
class CelsiusToFahrenheitAdapter implements FahrenheitTemperature {
    private CelsiusTemperature celsiusTemperature;

    public CelsiusToFahrenheitAdapter(CelsiusTemperature celsiusTemperature) {
        this.celsiusTemperature = celsiusTemperature;
    }
```

```
    @Override
    public double getFahrenheit() {
        double celsius = celsiusTemperature.getCelsius();
        return (celsius * 9 / 5) + 32;
    }
}

// Client Code
public class AdapterExample {
    public static void main(String[] args) {
        CelsiusTemperature celsius = new CelsiusTemperature(25);
        CelsiusToFahrenheitAdapter adapter = new
CelsiusToFahrenheitAdapter(celsius);

        FahrenheitTemperature fahrenheit = adapter; // Upcasting

        System.out.println("Celsius: " + celsius.getCelsius());
        System.out.println("Fahrenheit: " + fahrenheit.getFahrenheit());
    }
}
```

Explanation:

- FahrenheitTemperature is the Target interface.
- CelsiusTemperature is the Adaptee.
- CelsiusToFahrenheitAdapter is the Adapter. It implements FahrenheitTemperature and holds an instance of CelsiusTemperature. The getFahrenheit() method performs the conversion.

Benefits:

- **Reusability:** Allows you to reuse existing classes that have incompatible interfaces.

- **Flexibility:** Decouples the client from the Adaptee, making it easier to change the Adaptee implementation.
- **Open/Closed Principle:** You can introduce new Adapters for different Adaptees without modifying the client code.

When to Use:

- When you want to use an existing class, but its interface doesn't match the one you need.
- When you want to create a reusable class that can work with different, unforeseen classes.
- When you need to use several existing subclasses, but it's impractical to adapt their interface by modifying their source code.

Key Differences from Other Patterns:

- **Facade:** Provides a simplified interface to a complex subsystem. It doesn't change the interface of the underlying classes, but rather provides a higher-level interface.
- **Decorator:** Adds responsibilities to an object dynamically. It doesn't change the interface of the object, but rather enhances it.
- **Bridge:** Decouples an abstraction from its implementation. It's used when both the abstraction and the implementation can vary independently.

The Adapter pattern is a valuable tool for integrating existing code into new systems or for working with third-party libraries that have different interfaces than your application expects. It promotes code reusability, flexibility, and maintainability.

Explain how adaptee code is with incompatible interface

In the Celsius to Fahrenheit Adapter example, the incompatibility lies in the way temperatures are represented and accessed. Let's break it down:

- **Client's Expectation (Target Interface - FahrenheitTemperature):** The client code expects to work with temperatures in Fahrenheit. It has a

FahrenheitTemperature interface with a getFahrenheit() method. This method is designed to *directly* return a Fahrenheit temperature value.

Java

```
interface FahrenheitTemperature {
    double getFahrenheit(); // Returns Fahrenheit directly
}
```

- **Adaptee's Interface (CelsiusTemperature):** The CelsiusTemperature class represents temperatures in Celsius. It has a getCelsius() method that *directly* returns a Celsius temperature value.

Java

```
class CelsiusTemperature {
    private double celsius;

    // ... constructor ...

    public double getCelsius() { // Returns Celsius directly
        return celsius;
    }
}
```

The Incompatibility:

The core incompatibility is that the client wants a getFahrenheit() method, but the CelsiusTemperature class only provides a getCelsius() method. There's no way to directly use a CelsiusTemperature object where a FahrenheitTemperature object is expected.

Why not just change CelsiusTemperature?

In a real-world scenario, you often can't modify the Adaptee class. This could be because:

- It's a third-party library you don't have control over.

- It's an existing class in your codebase that's used in other parts of the system and changing it would have unintended consequences.
- You want to keep the Celsius representation separate for other purposes.

How the Adapter Solves the Incompatibility:

The CelsiusToFahrenheitAdapter acts as a bridge:

1. **Implements the Target Interface:** The adapter implements FahrenheitTemperature, so it *must* provide a getFahrenheit() method, fulfilling the client's expectation.
2. **Holds an Instance of the Adaptee:** The adapter has a CelsiusTemperature object as a member.
3. **Translates the Call:** Inside the adapter's getFahrenheit() method, it calls the getCelsius() method of the CelsiusTemperature object it holds, performs the conversion from Celsius to Fahrenheit, and then *returns* the result.

Java

```
class CelsiusToFahrenheitAdapter implements
FahrenheitTemperature {
    private CelsiusTemperature celsiusTemperature;

    // ... constructor ...

    @Override
    public double getFahrenheit() {
        double celsius = celsiusTemperature.getCelsius(); // Calls
Adaptee's method
        return (celsius * 9 / 5) + 32; // Performs the conversion
    }
}
```

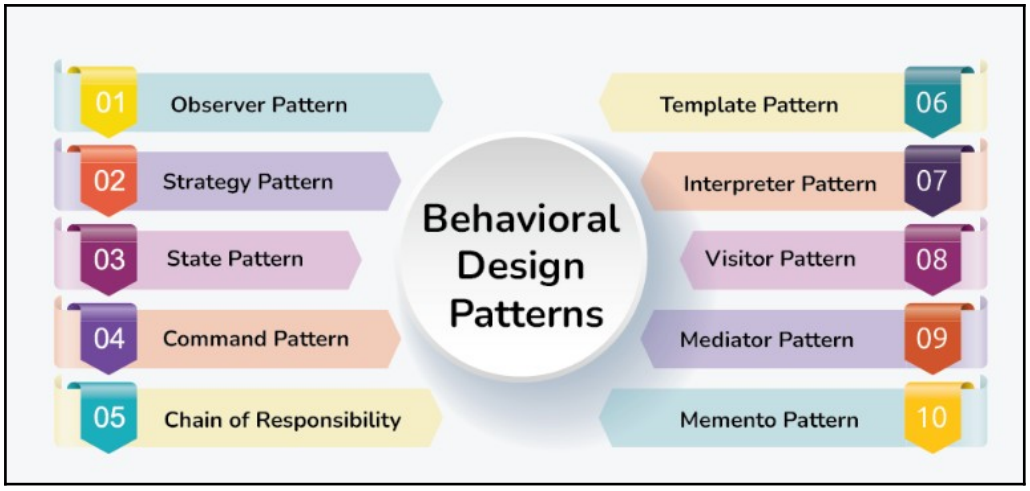
By performing the conversion within the adapter, the client code can now work seamlessly with Fahrenheit temperatures without needing to know anything about the underlying Celsius representation. The

adapter has effectively made the incompatible interfaces work together.

In summary, the incompatibility is not about the data *type* (both use double), but about the *interface* (the method names and the temperature scale they represent). The adapter provides the necessary translation to resolve this interface mismatch.

3. Behavioural Desing Pattern

These patterns identify common communication patterns between objects and realize these patterns.



1. Observer Design Pattern:

The Observer design pattern is a behavioral pattern that establishes a one-to-many dependency between objects. When the state of one object (the subject) changes, all its dependents (observers) are automatically notified and updated. It's also known as the publish-subscribe pattern.

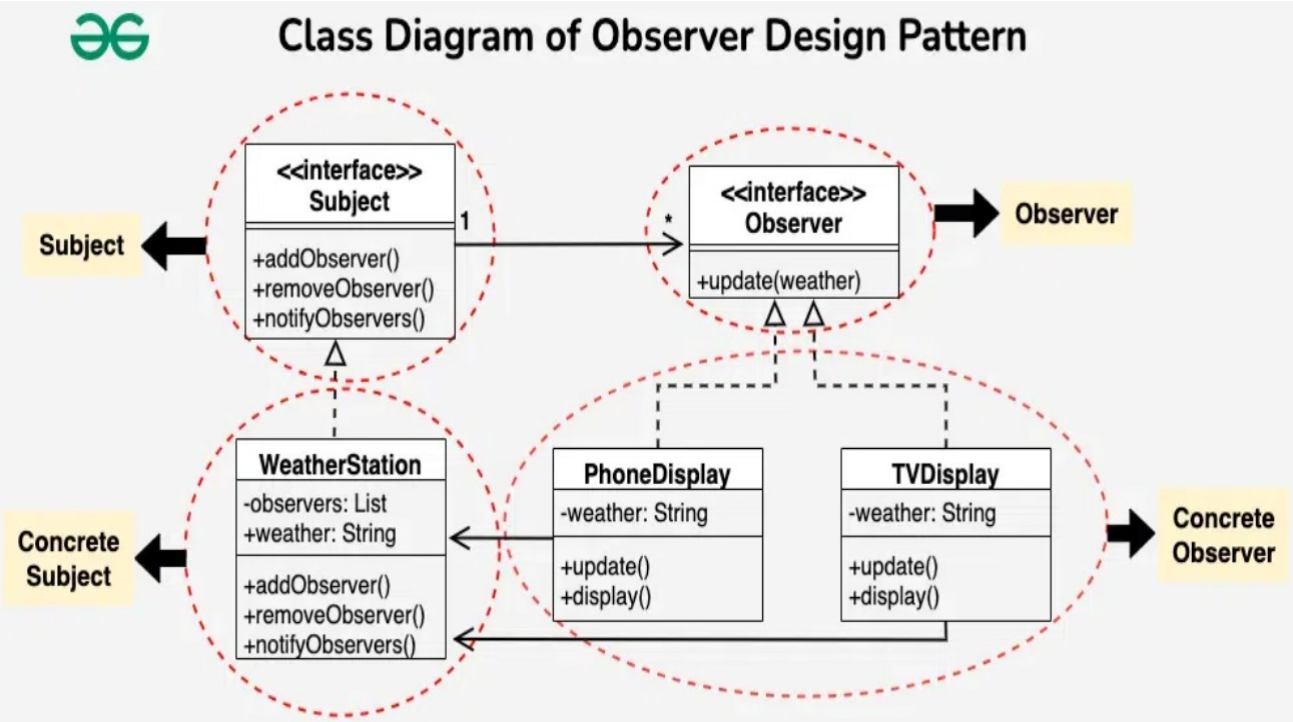
Problem:

Imagine you have an object (e.g., a weather station) whose state changes frequently (e.g., temperature, humidity). You have multiple other objects (e.g., displays, data loggers) that need to be updated whenever the weather station's state changes. If the weather station directly manages the updating of these objects, it becomes **tightly coupled** to them, making it difficult to add or remove observers.

Solution:

The Observer pattern decouples the subject from its observers. It defines:

- **Subject:** An interface or abstract class that provides methods for **attaching and detaching** observers.
- **Observer:** An interface or abstract class that defines a method for being notified of state changes.
- **Concrete Subject:** A concrete class that implements the Subject interface and maintains a list of observers. When its state changes, it notifies all attached observers.
- **Concrete Observer:** Concrete classes that implement the Observer interface and react to notifications from the subject.



Structure:

- **Subject:**
  - attach(Observer observer): Adds an observer to the list.
  - detach(Observer observer): Removes an observer from the list. This means that after detaching, the observer will no longer receive notifications from the subject when its state changes.
  - notifyObservers(): Notifies all attached observers.
- **Observer:**
  - update(Subject subject): Called by the subject to notify the observer of a state change.

Example (Weather Station and Displays):

Java

```
import java.util.ArrayList;
import java.util.List;
```

```
// Subject Interface
interface Subject {
    void attach(Observer observer);
    void detach(Observer observer);
    void notifyObservers();
}

// Observer Interface
interface Observer {
    void update(Subject subject);
}

// Concrete Subject (WeatherStation)
class WeatherStation implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private int temperature;

    public int getTemperature() {
        return temperature;
    }

    public void setTemperature(int temperature) {
        this.temperature = temperature;
        notifyObservers(); // Notify observers when temperature
changes
    }

    @Override
    public void attach(Observer observer) {
        observers.add(observer);
```

```
    }

    @Override
    public void detach(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(this);
        }
    }
}

// Concrete Observers (TemperatureDisplay, Log)
class TemperatureDisplay implements Observer {
    @Override
    public void update(Subject subject) {
        WeatherStation station = (WeatherStation) subject;
        System.out.println("Temperature Display: " +
station.getTemperature() + "°C");
    }
}

class Log implements Observer {
    @Override
    public void update(Subject subject) {
        WeatherStation station = (WeatherStation) subject;
```

```
        System.out.println("Logging: Temperature changed to " +
station.getTemperature() + "°C");
    }
}

// Client Code
public class ObserverExample {
    public static void main(String[] args) {
        WeatherStation station = new WeatherStation();

        TemperatureDisplay display = new TemperatureDisplay();
        Log log = new Log();

        station.attach(display);
        station.attach(log);

        station.setTemperature(25); // Output: Temperature Display:
25°C, Logging: Temperature changed to 25°C
        station.setTemperature(27); // Output: Temperature Display:
27°C, Logging: Temperature changed to 27°C

        station.detach(display); //Detach the display

        station.setTemperature(30); // Output: Logging: Temperature
changed to 30°C
    }
}
```

Explanation:

- Subject and Observer are the interfaces.

- WeatherStation is the concrete subject. It maintains a list of Observer objects and notifies them when the temperature changes.
- TemperatureDisplay and Log are concrete observers. They implement the update() method to react to notifications.

Benefits:

- **Loose Coupling:** The subject and observers are decoupled. The subject doesn't need to know the specific classes of its observers.
- **Open/Closed Principle:** You can add new observer types without modifying the subject.
- **Flexibility:** You can easily add or remove observers at runtime.

When to Use:

- When a change to one object requires changing other objects, and you don't want tight coupling.
- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to a subject may require an unknown number of other objects to be updated.

Real-World Examples:

- Event handling in GUI frameworks.
- Model-View-Controller (MVC) architecture.
- Spreadsheet applications (when a cell's value changes, dependent cells are updated).
- Notification systems.

The Observer pattern is a powerful tool for managing dependencies between objects and promoting loose coupling in your code.

2. Strategy Design Pattern

The Strategy design pattern is a behavioral pattern that lets you define a family of algorithms, encapsulate each one, and make them interchangeable. This allows the client to choose an algorithm at runtime without modifying the client's code.

Problem:

Imagine you have a class that needs to perform a certain task in different ways. For example, you might have a sorting algorithm that can sort data using different strategies (e.g., quicksort, mergesort, bubble sort). If the sorting logic is directly embedded within the class, it becomes difficult to switch between sorting algorithms or add new ones without modifying the class itself.

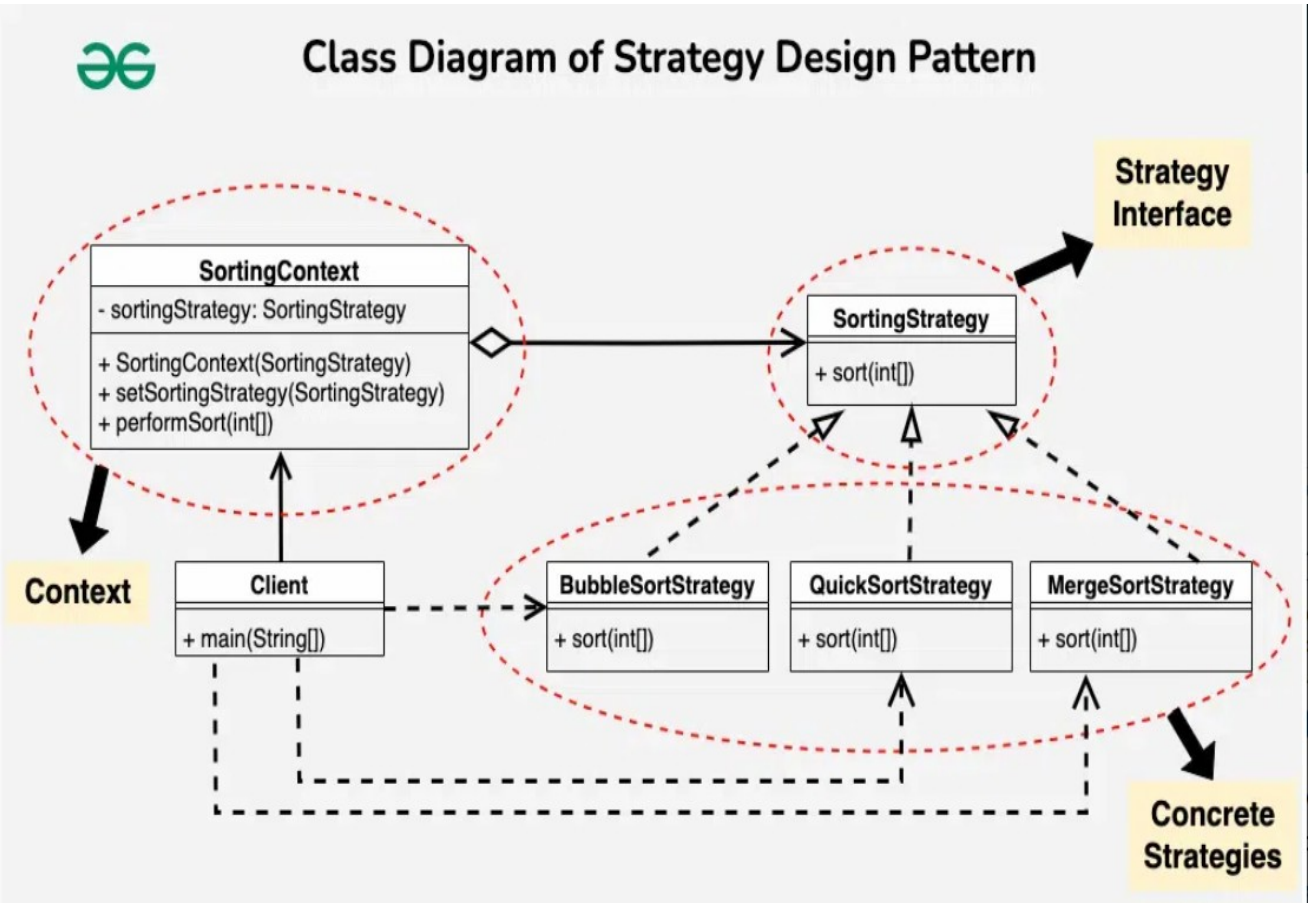
Solution:

The Strategy pattern solves this by:

- Defining an interface or abstract class for the strategies (algorithms).
- Creating concrete strategy classes that implement the strategy interface and provide the specific implementations of the algorithms.
- The context (the class that uses the strategies) holds a reference to a strategy object and delegates the task to the selected strategy.

Structure:

- **Strategy:** Defines the interface common to all supported algorithms.
- **Concrete Strategy:** Implements the Strategy interface, providing concrete implementations of the algorithms.
- **Context:** Holds a reference to a Strategy object and uses it to execute the algorithm.



Example (Sorting Algorithms):

Java

```
import java.util.Arrays;

// Strategy Interface
interface SortingStrategy {
    void sort(int[] array);
}

// Concrete Strategies
class BubbleSort implements SortingStrategy {
    @Override
```

```
    public void sort(int[] array) {
        System.out.println("Sorting using Bubble Sort");
        int n = array.length;
        for (int i = 0; i < n - 1; i++)
            for (int j = 0; j < n - i - 1; j++)
                if (array[j] > array[j + 1]) {
                    // swap arr[j] and arr[j+1]
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
    }
}

class QuickSort implements SortingStrategy {
    @Override
    public void sort(int[] array) {
        System.out.println("Sorting using Quick Sort");
        Arrays.sort(array); // Using Java's built-in quicksort for simplicity
    }
}

// Context
class Sorter {
    private SortingStrategy strategy;

    public Sorter(SortingStrategy strategy) {
        this.strategy = strategy;
    }
```

```
    public void setStrategy(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public void performSort(int[] array) {
        strategy.sort(array);
    }
}

// Client Code
public class StrategyExample {
    public static void main(String[] args) {
        int[] data = {5, 2, 8, 1, 9, 4};

        // Using Bubble Sort
        Sorter sorter = new Sorter(new BubbleSort());
        sorter.performSort(Arrays.copyOf(data, data.length)); // Create a copy to avoid modifying original

        // Using Quick Sort
        sorter.setStrategy(new QuickSort());
        sorter.performSort(Arrays.copyOf(data, data.length)); // Create a copy to avoid modifying original
    }
}
```

Explanation:

- SortingStrategy is the strategy interface.
- BubbleSort and QuickSort are concrete strategies.
- Sorter is the context. It holds a SortingStrategy object and delegates the sorting to it.

Benefits:

- **Open/Closed Principle:** You can add new sorting algorithms without modifying the Sorter class.
- **Flexibility:** You can easily switch between sorting algorithms at runtime.
- **Code Reusability:** The sorting algorithms are encapsulated in separate classes and can be reused in other parts of the application.
- **Avoids Large switch or if/else Statements:** Instead of having a large switch statement or nested if/else blocks to choose the algorithm, the logic is distributed among the concrete strategy classes.

When to Use:

- When you have multiple algorithms for a specific task and you want to be able to switch between them at runtime.
- When you want to avoid large switch statements or nested if/else blocks.
- When you want to encapsulate algorithms in separate classes for better code organization and reusability.
- When you need to choose between different implementations of an algorithm.

Key Differences from Other Patterns:

- **State:** The State pattern allows an object to alter its behavior when its internal state changes. The Strategy pattern focuses on choosing different algorithms to perform a task, not on changing the object's state.
- **Template Method:** The Template Method pattern defines the skeleton of an algorithm in a base class and lets subclasses redefine certain steps. The Strategy pattern defines entire algorithms in separate classes.

The Strategy pattern is a powerful tool for creating flexible and maintainable code when dealing with multiple algorithms or variations of a task.